

GPU v 0.93

Mathematica Setup (not relevant if you're reading the PDF)

Need to use a conditional on “printing to PDF” to hide this!

This writeup was all done in Mathematica. If you have access to Mathematica, you can download the companion notebook and look at the actual numbers, draw your own graphs from those numbers etc. But most people don't have Mathematica, so for you I've printed the notebook to a PDF.

If you use Mathematica, we need a way to paste results data from the command line apps into Mathematica.

Easiest solution appears to be

<http://szhorvat.net/pelican/pasting-tabular-data-from-the-web.html>

When you first open this notebook, say yes to “Allowing Dynamic Content”. You will need this to activate the two UI elements (“Show Input” and “Outline”) at the top of the document.

Next choose Evaluate Initialization Cells from the Evaluate menu (this will take a few tens of seconds to execute). This will load all internal variables (specifically all the many arrays of measurement data) into Mathematica, allowing you, if you want, to plot the graphs in different ways, or otherwise interact with the data.

Note the “Show Input” button at the top of this notebook; toggle it if you want to see the (sometimes copious!) input data for any particular graph.

The Mathematica code below adds that functionality to this notebook (not shown when “Show Input” is untoggled).

Also remember ctrl-clicking on a graph brings up a contextual menu, one of whose items, "Get Coordinates" is often useful in getting a quick, reasonably accurate feel for the coordinates of a point .

Introduction

I want to provide here a very quick summary of GPU issues, mainly to help situate anyone who reads this and wants to do further GPU investigation.

As always I start off with somewhat simplified explanations, then add in the complexity where I think it makes sense.

Suppose we have a very simple CPU, in-order, 1-wide, non-speculative. How can we make this “faster”? Well one path is the OoO, superscalar, speculative path discussed in great detail in Vol 1. But a second path is to create a throughput engine, one that is slow to generate a single result, but which can create many many single results in parallel. This is the thinking that leads us to GPUs.

So, given our simple CPU, suppose we have the following as a clear example problem: we have two large arrays (say ten million floating point values) A[] and B[], and we want to multiply them point by point and add the results to form a dot product (and we’re not too concerned with finicky details like the exact order of the multiplies and adds). If we write the obvious loop for our simple CPU, most of the time will be spent simply waiting: waiting for the loads from memory, then waiting for the next multiply to finish before we can perform the next add. But we can improve on this by splitting the problem into ten thousand *independent* subproblems, each of which corresponds to calculating the dot-product of two vectors 1000 elements long.

The question now is what’s the most efficient (ie smallest area and power) way to implement this?

- We want to run multiple of these subproblems at a time.
- We want to be able to switch to a different subproblem anytime our current subproblem has to wait on a DRAM load.

The most naive solution is just to create a whole lot of simple in-order CPUs, but that suffers from at least two problems. It’s not area efficient because most of the time most of the CPUs are just waiting for memory. And it’s not energy efficient because each of these CPUs is executing the same stream of instructions, but we’re not using that fact to save energy.

Parallelism in time - fixed

To do better, let’s remember a proposed CPU design from the 80s or early 90s called a *Barrel Processor*. The idea of this sort of processor is that it holds multiple copies of state, and every cycle it switches to a different copy. So if there are eight copies of state, we begin executing the instruction (the “thread”) associated with state 0, next cycle we switch to state 1, and so on. By the time we get back to state 0 eight cycles have passed and hopefully the desired result (a load from cache, or an FP multiply or whatever) has been calculated and can immediately be used by the next instruction in the thread. We’re executing eight threads in parallel; each thread runs at one eighth the speed, but overall we get better throughout. By providing this fixed delay between successive instructions of the same thread, we can avoid many of the issues that we try to fix in a CPU via speculation, caches, OoO processing and

similar such (very area-expensive) machinery. The price we pay is that this only works if we want a throughput engine (execute many small sub-jobs) not a latency engine (execute one job that isn't easily split into pieces)...

You can see a barrel processor as somewhat like SMT, implemented on a very simple CPU that can only execute one instruction at a time (so the “multi-threading” part of SMT is there, the “simultaneous” part is more a question of definitions in terms of what you consider simultaneous in this simple pipeline).

Imagine we upgrade our very simple CPU to a Barrel Processor and give it, say, eight copies of state. The primary cost to this is replicating the register set eight times. We now split our dot product task into eight threads defined by something like

- a few instructions (basically some setup, then a loop over ten million/8=1,250,000 iterations)
- a thread ID (0..7) and a per-thread stack
- some startup code at the beginning that does something like calculate the array addresses

```
Alocal=A+125,000*threadID, Blocal=B+125,000*threadID
```

We can now execute these eight (identical except for the thread ID which is used to initialize the source data) threads barrel-style on our very simple CPU and instead of spending about 7/8 of our time waiting for an operation (L1 load or FP operation) to complete, we're basically doing something useful every cycle.

Finally at the end we have some minor cleanup to add together the eight sub-dot-products calculated by each thread.

We have added some degree of complexity to our CPU (the register state and something that switches the instruction stream/threadID each cycle) but much less than would be required to get this performance by adding OoO functionality.

This is just a start. Don't yet consider the many possible problems (most notably to do with memory); let's see where this thinking goes.

Parallelism in time - flexible

Note also that we've done this by moving the “support for multiple threads” into the CPU. This is not eight independent cores with an OS deciding which threads execute where and when; instead we have “thread manager” hardware sitting on top of the CPU, with a local memory storing the list of threads (eg thread number, stack pointer, and PC for each thread) and each cycle moving on to the next thread that gets its instruction sent into the CPU.

The example we gave (dot product) assumed all eight threads were identical code. But that's not essential; we could have eight (slightly, or even very) different instruction streams registered with the thread manager.

Next, the simplest scheduler simply increments the threadID each cycle, but once we have different instruction streams, one stream may be doing low latency operations (lots of integer arithmetic, result available the next cycle, while another is doing lots of memory access and a third is doing lots of FP

work).

So we can start to add some intelligence to the thread manager to know (or at least predict) which threads will have their results available in the next cycle vs which will not, and schedule the next thread based on this fact. Now we're retaining the idea of multiple threads, managed by hardware, but we're giving up the precise clock-work cycling of the barrel processor for more flexible scheduling.

This is our very simple prototypical GPU. It has

- one (simple, many-stage) pipeline,
- many register sets (each set associated with a different thread and only accessible by that thread),
- some simple score-boarding (to track which threads have or have not yet generated their results, and so can move on to their next instruction), and
- a scheduler to choose the next thread to execute each cycle.

- threads cannot interfere with each other via register space, but can (in principle) overwrite each other in address space.

We have no MMU/TLB, no virtual addresses, and the assumption is that all the threads belong to the same process, so they have some degree of co-ordination (via writing the code correctly!) and it's no security issue if one reads what another is doing.

- in the initial days of GPUs, the assumption was that the GPU would, at any one time, be executing a single task of a large number of essentially identical threads. Nowadays we assume the GPU will be executing a (smallish) number of tasks, each of a large number of essentially identical threads. I mention this because once we start to discuss scheduling and resources below, you always want to keep these two conditions in mind. On the one hand, the “default” condition, historically and in terms of ease of understanding, is executing a large number of very similar to identical threads; on the other hand the modern condition is executing a mix of threads, of which say half or a third are all identical, but different from the other half or thirds.

The assumption is that code that fits this model (like a dot product, or piecewise multiplication of two large arrays) can fairly easily split into separate pieces the memory addresses into which it writes, so that the different threads will not overwrite each other and, in fact, don't even care about their relative order of execution. If you need more precise ordering, you need to insert specific barriers of some sort into the code, to make sure that, eg, thread 0 has reached a specific instruction (and so written out all its data) before thread 1 can begin. Much of the interesting smartness now lives in the Thread Scheduler which needs to choose an optimal thread each cycle to insert into the pipeline to ensure that the pipeline never blocks.

initial scheduler considerations

We will grow this GPU soon, but right now let's consider some aspects of what we have already defined. Think about what's needed for a smart scheduler to do its job. The simple round robin barrel scheduler incremented a threadID each cycle, but could schedule a thread that cannot execute (eg because its

previous instruction was a load that missed to DRAM instead of hitting in cache, and the delay is many more than just eight cycles). So the cycles while that thread is waiting for DRAM, but gets scheduled, are lost as empty bubbles. We can similarly lose cycles if a thread has missed in the I-cache.

So we want a smarter scheduler. But that means spending some area. How much do we want to spend? Your first thought might be to just skip over the threads in the rotation that are waiting for memory. But that may throw off the timing of all the other threads which implicitly assume eight cycles between execution to generate a result!

The most practical, low-area, solution is probably a two level solution: we have a pool of registers to support say 16 thread contexts, and we maintain a set of 8 “active” threads. When a thread misses to DRAM, its threadID and PC are moved out of the active set, and one of the inactive threads (for definiteness let’s say the oldest) is moved in. The 8 active threads operate as a fixed pattern barrel processor. This gives us efficiency at low additional complexity – but now to fully use the machine we need to split our problem into 16, not just 8, pieces...

Alternatively we can go to the other end of the spectrum, to a much more sophisticated solution.

- assume eight mini front-ends which implement Instruction Fetch, Decode, and Hazard Detection (which is essentially the simple in-order CPU version of Rename/Allocate, and tests that the two inputs into the instruction are either already available or will be available by the time the instruction requires them).
- these eight front ends all route to an arbiter, which now has the information it requires to send one of the decoded instructions on to the main pipeline.

This scheme allows any thread to execute as soon as its data is ready, without having to wait a fixed eight cycles even if the previous instruction was something lightweight and taking only one cycle. In principle it allows threads that are executing a lot of low latency instructions to finish earlier. Of course it uses more area.

Between these two extremes: it’s sub-optimal to have eight front-ends that are each doing nothing most of the time (every cycle only one of the eight will have its instruction chosen so that it can move forward and execute another fetch).

It might be more efficient to have say just two front-ends, each of which is a four-way barrel processor, so that the front-end can cover I-cache latency and a delay of a few possible cycles while we wait for a hazard on one of the threads to clear. The two front ends each generate (via whatever mechanism) a potential instruction for the next cycle, and the arbiter chooses which of the two to execute.

There are many alternative ways to slice this up; but you can see the general scope of the issue: for each of the threads known by the Thread Manager

- we can have delays in I-fetch (from the I-cache, perhaps from the L2 cache, perhaps from DRAM)
- we probably want to move threads waiting on DRAM into some sort of secondary storage so that we aren’t, every cycle, burning energy considering “should we schedule this thread?” and getting the answer “no!” for hundreds of cycles

- after decode we can then have a delay (usually a few cycles, but sometime much longer if we are waiting for DRAM data) and frequently variable (eg both register access and L1 cache access may suffer from bank conflicts)
- before the instruction is executable (ie all hazards cleared)

So in a perfect world we want something like

- two pools of threads, a “long term parking” pool and an “active” pool; and
- + two “decoupling points” (buffers between a pipeline stage; at Fetch; and at Hazard Detection) where we can move an instruction into some sort of short-term “wait until activated” storage), while minimizing the amount of duplicated logic, perhaps down to something as simple as a single pipeline that looks something like

- Instruction Fetch (send request to the cache/memory system)
- + decoupling point where we switch between threads so that some can be waiting for cache misses
- move fetched instruction to Decode
- move fetched instruction to Hazard Detection and Register Access
- + decoupling point where we switch between threads
- rest of the pipeline

Now (ideally! and assuming a large enough pool of threads...) we will never have to wait for instruction cache misses, data cache misses, sequential instruction dependencies, or anything else; we can always find a thread to execute.

Everybody follows a model more or less like the above, but the details are surprisingly varied.

nVidia move a lot of the work to the compiler; they provide scoreboard hardware for instructions like loads that are pretty much certain to have variable latencies, but rely on hard-coded latencies (enforced by the compiler) to ensure that say a subsequent FMAC instruction is not executed before its required arguments have been calculated by earlier instructions (ie, to simplify, each instruction will have a count associated with it, telling the scheduler that this instruction cannot be issued until at least n instructions after the previous instruction in this thread). This saves area, but probably means they have to assume worst case scenarios for possibly variable latency (eg latency of accessing registers in the face of register bank collisions).

It's hard to be sure, but Intel and Apple seem to be rather more on the dynamic side of things, using more of the superscalar scoreboard machinery they both use for their CPUs, though in much simplified form.

This is still not the end of the story.

Real GPUs have some specialized hardware for particular tasks that are complicated to do in SW, for example texture lookup. Assume we have a texture lookup unit that is not pipelined and takes four cycles. How does this interact with our scheduler?

In particular, suppose that we schedule two texture lookups back to back. The first will start to execute in the texture unit, the second will block the pipeline because the texture unit is not pipelined; it can't accept a new instruction until the current one is done. This is not exactly a Hazard (correctness

issue, with a dependency on the results of a previous instruction) and will not be picked up by our Hazard scoreboard; it is more a performance issue of bandwidth. Across the pool of active threads, the scheduler should not submit a texture instruction more frequently than once every four cycles; though if we do things will work correctly, we'll just lose a few cycles in waiting.

There are in fact multiple versions of this sort of thing. Another is that GPUs have a variety of different storage locations (for example a *Scratchpad cache* or a *Constant cache*, terms to be explained later) and these may also have different bandwidths, so that the Scratchpad can perhaps accept one access every two cycles, and the Constant cache one read every four cycles).

One way to handle this is “precisely”. In the “second scheduler”, the one tracking Hazards, we implement something like a set of counters tracking the bandwidth of each problematic item.

Alternatively we can label the threads given to the GPU with various flags (this one makes heavy use of texture, that one makes heavy use of the shared cache) and the “first scheduler”, the one tracking which instruction gets the next PC and thus accesses the Instruction Cache, choose threads based on these flags. This scheme won’t be as precise, but will use less energy and area, and may be good enough for everything to average out in the end?

Intermediate between these two options (and what Apple seems to do) is to pre-decode instructions as they flow in from L2, so that extra bits associated with each instruction describe whether these bandwidth-constrained units are used.

It’s interesting to see where different vendors concentrate their efforts. The public documentation from nVidia or AMD do not seem to bother much with this issue of bandwidth to various constrained units, whereas its something Apple is constantly tinkering with.

So at this stage we have a GPU core that switches every cycle between one of some number of threads (perhaps 8 active threads from a pool of 16 thread contexts). And we’ve seen that scheduling is a non-trivial problem, constrained by

- misses to DRAM (which remove threads from the pool of active, schedulable, threads),
- sequential dependencies (
- + The simple version: a given thread cannot execute its next instruction until the previous one is complete
- + The complex version: a given thread cannot execute its next instruction until its dependencies are available, but could execute an independent next instruction...), and
- execution unit bandwidths (we want to interleave instructions that utilize different special purpose units of various sorts, and different storage areas).

resource considerations (registers and scratchpad)

So far in our design we’ve just assumed that our barrel processor offers a given thread some number, lets say 32 registers, and replicates that to say $8 \times 32 = 256$ registers to handle the switch to a new thread each cycle. This matches how eg x86 SMT gives each CPU thread (aka virtual core) a full set of registers. But this is sub-optimal. If we have 256 registers available and my threads only use say 10 (or fewer) registers, then maybe it might be nice if I could pack in three times as many threads, and have a higher

chance of always having a thread available to run in spite of cache misses and suchlike? This requires the GPU Thread Manager now to be able to track 24 rather than 8 threads. In other words rather than 8 thread contexts (threadID, PC, set of registers) I now need 24 such contexts (threadID, PC, and some sort of mapping from the 10 registers I use into 10 registers placed somewhere in the pool of 256 registers).

The real cost to all this machinery is the register storage, not the threadID and PC storage; so lets try to get maximum value from that by allowing for more threads if each thread is light in register usage!

Modern GPUs do this by

- each thread indicates how many registers it uses, and
- at “configuration time”, ie when the OS+library code is setting up all the threads that will execute on the GPU until the current task is done, the library code tries to pack together as many threads as possible subject to not overflowing thread storage. So our design, as described, could pack knowledge about from 8 to 24 threads into the Thread Manager at configuration time, depending on how many registers each thread uses. Each thread refers to its registers as say r0..r10, but behind the scenes a register number remapping is setup so that for say thread 0 r0→p0, r1→p1, and so on; for thread 1 r0→p10, r1→p11, and so on. This setup (known threads and how their registers are mapped) persists until the threads have all finished executing, at which point we accept the next command buffer and reconfigure. During execution the actual mapping of the thread’s register, r# to a physical register, p#, can happen at any convenient stage, for example perhaps during Decode.

This seems obvious enough and a good step forward, but note some consequences. How exactly do we perform this remapping (a full remapping table is most flexible, but much cheaper is just an offset to where r0 is placed for each thread. A simple offset value works fine as long as our threads are identical and don’t change. But if we start interleaving threads executing different code, then we can land up with fragmentation, ie two unused blocks of 16 register are available, but the new thread I want to start using requires 20 registers...)

The other primary resource whose usage needs to be handled is Scratchpad.

The historical trajectory is that the first GPUs did not have caches, or had something like a read-only L1. But this means that if threads want to communicate with each other, they have to do so by reading and writing to DRAM, which is obviously very slow. Thus the GPUs were provided with Scratchpad storage, some local on-chip storage (think say 16kB) forming its own address space unrelated to the primary address space, and thus with no constraints regarding coherency, and with whatever memory model was convenient. In the early days when every thread was part of a single co-operating process, that process would carve up and use Scratchpad storage as it wished; but once multiple independent agents began to share the GPU some sort of co-ordination was required. We’ll see the details of how this works later, but you could imagine Scratchpad now as consisting, conceptually of something like say 64 blocks of storage each 1kB in size. A given set of interacting threads can request some amount of this storage up to some limit (eg 32kB) and will see its allocation as a contiguous address space from 0..limit. Just like we try to pack in multiple threads up to the limit imposed by the total register usage of

all threads, so we try to pack in multiple threads up to the limit imposed by the total Scratchpad usage of all the sets of co-operating threads.

As an aside, Scratchpad storage is basically SRAM associated with a core. In other words, apart from some logic surrounding it handling things like addressing, it looks a lot like cache. This has led to many varied design choices over the years.

The first evolutionary stage tended to have separate Scratchpad storage along with a pool of read-only storage to be used by read-only clients (eg texture cache and a constant cache). Maybe these were separate caches, maybe they shared the same cache.

The next stage was to allow a per-core cache to also capture stores (ie writeback rather than write-through cache). At this point you can

- unify the L1D with Scratchpad, which has the advantage that you can, somewhat dynamically, use as much of this storage as required for Scratchpad and all the rest for L1D, with no waste. Disadvantage is that you have to provide this storage with enough bandwidth to service both sets of clients, both Scratchpad load/stores and “main memory” load/stores.
- alternatively you can separate L1D and Scratchpad as different physical pools of storage, but try to get additional value out of the Scratchpad storage. This is what Apple has done so far, using a single pool of SRAM as some combination of Scratchpad storage (for threads that want that), behind-the-scenes Tile storage (for the TBDR graphics functionality of the GPU), and visible Tile Shader storage (for developer shaders, ie threads, that wish to manipulate the output results from the graphics system, eg adding shadows to a scene).

You could in principle also roll the read-only caches (like Texture and Constant) into this pool of storage, but everyone seems to feel that the simplicity (ie smaller area) and increased bandwidth available from keeping these separate is worth maintaining.

As additional functionality is added to GPUs, additional resources may also be added. So far the Tensor functionality (essentially matrix multiply functionality) added by Apple, AMD, and nVidia has used the standard thread registers as storage, but more aggressive support for neural functionality might result in some sort of dedicated local “layer” or “weight” storage. Likewise one could imagine new storage provided to support ray tracing functionality.

Parallelism in space - SIMD/lanes

At this point we have a design that, with some extra storage but without much extra logic (mainly the scheduler[s]) will, with enough threads doing enough different things, hopefully never block, not on any sort of cache miss, not on instructions dependent on earlier instructions, not on execution unit bandwidth (eg texture unit can only generate a new output every four cycles).

Our simple CPU/GPU has gone from generating a result perhaps once every six cycles on average to a result every cycle.

At this point how can we make it even more performant?

Conceptually imagine that we make all the registers NEON type registers, and likewise all the execution units (FP multiply, integer add, etc) NEON type units. So now one instruction describes what we want to do to, say, four data units at once and we achieve four times the throughput. This works if the work we want to do is essentially identical, on elements close together in storage, (like our dot product example) which is usually true. But there are two problems.

- The first is branches. We want to be able to write/compile code that looks like standard code, with statements like

```
(if loaded value >0) do {} else {}.
```

This can be handled with predicates, the same way AVX512 or ARM SVE use predicates, to execute some lanes of a SIMD computation while not executing others. The HW and compiler of a GPU make this more hassle-free than SIMD on a CPU (eg handling things like nested if statements) so we can just take it as given that this is more-or-less easily handled.

The second problem is less obvious, but more important, and has to do with loads (and, to a lesser extent, stores). A SIMD design on the CPU involves loading say 128b or 256b of *contiguous* data into a NEON/AVX register, but some of the use cases for GPUs load their data from non-contiguous locations. Suppose, for example, that our current block of code is supposed to run through an array of ten thousand polygons, test something about each one (eg visibility) and write out the list of visible polygons. This fits our initial barrel processing model (eg define eight identical threads and give each one a reference pointer to one eighth of the array) but fits the NEON model much less well in terms of the data load, since some of the threads will be loading triangles, some quads, some pentagons etc; the data access is irregular, not nicely shaped like NEON code expects).

For this reason we expand the GPU “in space” via something that looks somewhat like NEON or AVX SIMD but is not exactly the same.

- Conceptually we execute the same (possibly predicated) instruction across multiple lanes, like SIMD

- Conceptually we have wide registers consisting of a single scalar value per lane, like SIMD
- But we have more load/store flexibility into these registers, so that each lane of the register can be loaded/stored to a very different address. (SIMD scatter/gather kinda sorta does this, but in a less elegant way.)

- Simple conditionals (if/then/else) are handled by predication. The compiler and HW do this behind the scenes, including nested if/then statements, so the developer doesn't have to worry about it. The HW to do this is masks of predicate bits that are pushed and popped on a stack – kinda obvious and uninteresting.

- In principle you could execute indirect function calls. The most obvious way to handle this is via a predicate mask that masks one lane at a time, so now you're only executing $\frac{1}{32}$ lanes each cycle, and,

--
yes, you're losing a lot of efficiency!

A smarter design might at least try to aggregate all cases that route through the same function address so that those lanes run simultaneously. But overall procptr/virtual function style programming is not recommended for GPUs!

The language of GPUs becomes intensely messy at this point. The language I will use is that

- a thread is a sequence of SIMD-like instructions. Threads alternate in time across a GPU core as scheduled by a Thread Manager
- a lane is one of N identical pipelines (in our example we set N as four, in a real GPU everyone seems to have settled on N=32 as about the best choice) that all execute the same instruction in a given cycle, but on different, per lane, data.

The set of N identical pipelines and the code executing on it is variously called a *warp* (nVidia), a *wavefront* (AMD) or, by Apple, a *SIMD group*. Many people discussing the Apple GPUs then refer to this set of N identical pipelines not as a SIMD group but as a “SIMD” which is somewhat confusing until you know what they are talking about.

GPU Marketing, in the usual quest for bigger numbers no matter how misleading, then tends to call the product of the number of threads (independent PCs that the Thread Scheduler juggles) times the number of lanes the “number of *marketing threads*”. This is intensely unhelpful understanding anything, and I will never do this. I will usually refer to threads and lanes, or SIMDs/warps and lanes.

At various times the optimal way to split these concepts has been all over the place, with both AMD and Intel for example having multiple lanes of execution, each lane then executing a wide SIMD instruction; for example you might have sixteen lanes each executing 4-wide SIMD. Intel, in particular, took this to crazy levels of complexity. But as of 2023 pretty much everyone has calmed down and settled on essentially the model I am describing.

So our model has now become a GPU Core consisting of a barrel-like or SMT-like processor (choose one, from a collection of 8 or so) threads to execute each cycle, where each thread executes an instruction that applies to a 32-wide SIMD-like register.

Parallelism in space - “quadrants”

There is one more level of organization. A single “GPU Core” consists not of a single set of 32 lanes, but of four independent such sets.

Why would you do such a thing? It boils down to “how much is most efficient to share”?

By doing things this way each of the four quadrants can share at least:

- some of the Thread Manager and high level scheduling hardware
- perhaps some required but rarely used specialist graphics hardware
- the L1 and scratchpad caches

For example if we assume that, on average say, about 20% of GPU instructions are loads and stores, then by funneling four mini-cores through a single cache, then the cache bandwidth is being used on most cycles, rather than being unused most of the time.

I think it's most useful to think of this as four independent GPU mini-cores; (in the case of Apple each handling 6..24 threads), and which happen to share various caches. Usually this cache sharing is not an issue, but in principle it could be for certain types of code that consist of almost nothing but data movement.

Once again this is a design that has been converged to over a decade of multiple variations on this theme. At times a core has been split into as few as two or as many as six mini-cores sharing the same caches, but everyone seems to now agree on four as optimal.

Some people seem to think of this design as

- a core is “four wide” so each cycle it can, in theory, dispatch four instructions to four independent “units” that are processing these 32-wide NEON or AVX512 instructions.
- there's a single L1I feeding this core and a single high-level thread manager that knows about the $4 \times 6 = 24..4 \times 24 = 96$ threads that are currently in play (either active or pending).
- there are four lower level schedulers that decide each cycle which of the active subset of these 24..96 threads to execute on each of the four “wide units”.
- there is (at least conceptually) a single L1D shared across the entire core.

Mostly I see this viewpoint as more confusing than enlightening; I think considering each quadrant as the fundamental “core-like” unit is most helpful.

I mention it mainly so that, when you encounter something that talks this way, you have some idea what they are talking about.

It's unclear the extent to which the above view point is valid. For example

- Is there a single core-wide pool of registers, or four separate (and separately managed) pools of registers? Unclear! As far as I can tell, other vendors provide a per-mini-core pool of registers that are *not* shared. Compare this with Scratchpad storage which *is* shared across the whole core.

New Apple (ie A17/M3) definitely have a single pool of register storage (to be discussed much later). As of M1/M2 the register pool may or may not be shared.

- Can execution of a given thread move from one quadrant to another? Once again on other vendors this is clearly not the case; for Apple again it's unclear.

When we start to look at the history of GPUs, nVidia has the best diagrams and the clearest history, so it's very tempting to shoe-horn any other design into assuming it's a variant of nVidia. This may be a mistake.

The (current, after some evolution) nVidia design is essentially a

- per-core L-cache
- warps are *statically* assigned to one of four quadrants
- each quadrant has a lightweight instruction sequencer that pulls a few instructions from the L-cache into a quadrant buffer that holds a few instructions for a few warps (think L0 cache) and, each cycle, decides which instruction to execute next from the pool of available instructions in the buffer which are runnable (ie do not depend on a not yet complete prior instruction). Each mini-core is mostly a self-contained world with its own buffer, sequencer, registers, and execution units. Through a non-obvious piece of timing trickery, each mini-core is often able to run two instructions per cycle even though apparently only one instruction per cycle is dispatched from the instruction sequencer.

Now consider an alternative design where we have

- per-core L1-cache
- a single pool of pending warps, active warps, and registers
- a single high-level instruction sequencer that sends traces (straightline branch-free sequences of code from 1 to 16 instructions long, but clustering around say 8 instructions long) to execution units. You can think of these traces as something like a single super-instruction that will execute over multiple cycles one instruction at a time. These traces are called *clauses*.
- four FP execution units (each 32 lanes wide), four load execution units, four texture execution units, etc; each with their own small L0 cache in front for holding clauses, and their own simple sequencer capable of finding a clause in the L0 cache by clauseID, then walking along it one instruction at a time.

Finally we assume that like instructions (loads, texture, FP) are clustered together by the compiler as much as possible so that a clause is a load clause or an FP clause or a texture clause, ie a run of instructions of just one type, and that this fairly well matches actual GPU code, so that these traces are fairly long, say on average at least four instructions long.

The version of this design that clusters instruction types together is what AMD called clauses (back in the day when they used use clauses). It has obvious advantages in running multiple instruction types simultaneously because you only need to make a high level decision every few cycles about which clause executes next from a given warp, you don't have to make say three cycle-by-cycle scheduling decisions as to what instruction gets executed by which execution unit.

A different version of this idea was used by ARM GPUs (eg the Midgard/Bifrost line). These GPUs don't require that all the instructions of a clause be of the same type, but as I understand it, they do treat an entire clause as a single scheduling unit. The main consequence of this is that a single clause can have one memory load instruction, and long-range dependency tracking (so that you know when a dependent clause can start execution based on completion of an earlier instruction) can track only clauses, not individual instructions. However this scheme doesn't obviously get you the "natural" simultaneous use of load/store, datapath, and special function units that you get from the AMD scheme.

Yet a third related idea is that Imagination's 2025 (E-series) GPUs utilize a concept they call *burst processing*. Imagination's ideas are specially interesting in the context of Apple, because the two companies still seem to work very closely together, somewhat like the ARM/Apple relationship through even closer.

Burst processing appears to be the constructing of sequences of instructions somewhat like a clause, with the significant feature being that instruction intermediate results are not stored in the register file/operand cache but in some sort of very local storage (presumably a few registers built into each ALU). This allows saving power, and register bandwidth, by not having to move back and forth temporary results that will soon be discarded. Of course, for this to work the instruction burst has to be non-uninterruptible (since otherwise those intermediate temporaries will be lost) which in turn means you can't swap out this execution for a different warp, even if you're blocked for a cycle or two waiting for a result to be calculated. So can the compiler usually schedule bursts that are long enough to be useful, but that don't have to wait for results within a burst?

This seems like an interesting idea, at least worth exploring. It's worth noting that Apple have a recent patent (2024) <https://patents.google.com/patent/US20250103292A1> *Matrix Multiplier Caching*, dis-

cussed in volume 8, that provides a simplified version of this, allowing for one temporary storage register in each FMA unit. This may be a tentative first step towards the full Imagination idea, or it may be that Apple's simulations show that for their design, pretty much all the value of this idea resides in continual reuse of the output of the FMA unit.

The nVidia design is fairly easy to model in one's mind at a cycle by cycle level.

The second design is much more distributed design and harder to track. One can see how it would work at a high level

- the outer-level sequencer switches from one warp to another to another, each cycle, for one warp codestream, detecting the branch points (so detecting the straightline runs of code, and instruction types) and sending the detected clause to the appropriate L0I cache (load L0I, texture L0I, FP L0I, etc)
- each of these L0I's has multiple banks (say 6 to 8 banks) so that, in any cycle, each of the four connected units (eg four FP units) can each load the next instruction generated by their sequencer and probably not hit a bank collision
- obviously there are many details being avoided here in terms of handling dependencies within clauses (instruction FP2 depends on instruction FP1, so the FP unit that executes FP1 should switch to executing a different clause until FP1's result is available) and across clauses (FP1 depends on an earlier load which may have missed to DRAM) but conceptually you can see how the scheme might work.

On average, then, if we are generating one clause per cycle, and a clause is four instructions long, then more or less on average we execute four instructions per cycle, so it looks like a single core is more or less running four execution units (say FP32 units, each 32 lanes wide) per cycle, so superficially this looks like four quadrants nVidia style.

But if clauses are longer, so on average we are generating say clauses of 12 instructions per cycle, and the clauses cover a wide variety of different execution units (load, texture and FP), now it starts to look like we're not just four quadrants, but that each quadrant is able to execute three instructions per cycle, an FP, a load, and a texture instruction. This is, in some sense, true, but it's true because the clauses are on average 12 instructions long, and because we have so many small distributed instruction sequencers associated with each execution unit. This is very different from the nVidia model of four quadrant sequencers, which would each be looking for three instructions per cycle to send to three different execution units within a quadrant.

How much of the Apple design is like the above?

Apple definitely uses clauses, as described (short straightline code, instructions are one of a few classes) likewise for L0 caches; and distributed instruction sequencing (a high level version shared by the entire core that handles control flow [eg branches and function calls] for each warp, to generate a sequence of clauses; and low level instruction sequencing that walks a particular clause and occasionally switches clauses as hazards [dependency on an earlier instruction] require).

What is NOT clear is whether they use a single pool of registers, a single set of L0I caches, and a single set of execution units; or whether these are grouped into four independent buckets we can call quadrants.

It may even be that earlier designs (say M1 and M2) split functionality into quadrants, and it is only with M3 that we utilize a single pool of registers and L0I caches. A single pool design is better for load balancing, but is more complex.

I know this seems overwhelming on first reading! As you read below, you'll first see evolution of the nVidia designs (somewhat easy to understand) then later the details of the Apple clause scheme which should make things more clear.

All this, then describes a single GPU, something (details vary) looking like

- an L1I cache, an L1D cache, Scratchpad storage, some specialized graphics caches and hardware
- (either actually, or effectively, looking from the outside) four quadrants of 32-wide execution lanes each with its own scheduler and register sets and
- over short runs of time (as rapidly as one cycle to the next) execution units alternating execution between threads, as determined by the scheduler.

Once you have such a GPU core that you like, you can then replicate it multiple times. You generally want all these GPU cores to share a (basically traditional, read/write) L2 cache, both for usual data reuse reasons, and for communication between cores. You replicate cores (and grow the L2) as large as you can until everything becomes unwieldy, at which point call your GPU a “chiplet” and place two side by side, with some sort of cache coherency protocol keeping the two L2’s coherent.

This is essentially where we are today at the highest end. nVidia’s largest designs are physically a single piece of silicon, but internally split into two GPUs with coherent L2s but a single pool of on-package DRAM; likewise for AMD’s MI250; and for Apple’s M2 Ultra. Everyone looks poised to, as the next step, expand this to say four “chiplets”.

nVidia and AMD, meanwhile, also expand their coherency protocol off-chip via InfinityLink or nvLink, to grow what you might think of as “NUMA-like” GPUs, where the GPU’s are all mutually cache coherent (and see a single view of memory), but the two that are placed on a single chip can communicate with each other and their local RAM much faster than across these links.

Kernels, threadgroups, and warps, oh my

Let’s now get slightly more accurate. So far we’ve assumed a single task (a *kernel*) is split into threads executing 32-wide SIMD. There is an intermediate level of organization that we have not yet discussed:

Kernels are split into *threadgroups* which consist of individual threads of SIMDs.

Threadblocks tend to be from about 16 to 32 threads/warps/wavefronts/SIMDs in size (ie about 512 to 1024 lanes in size).

Again terminology will drive you insane.

- Everyone (pretty much) calls the largest level of task a kernel (or a grid).
- A kernel consists of *threadblocks* (nVidia) or *workgroups* (AMD) or *threadgroups* (Apple) or *CTAs*, *Cooperative Thread Arrays*, used everywhere in the academic literature and nowhere else. A better name (in my terminology) would be a lanegroup, but nobody uses that term.

- These threadblocks/threadgroups consist of *warps/wavefronts/SIMDs/threads* which execute 32-wide *lanes*.

It will take some time to get use to these terms, but just be patient; like learning any foreign language it starts off being hard work until, one day, it's just obvious and natural.

The point of a threadblock is that it's a collection of threads that are *guaranteed* to be *scheduled together on the same core*.

Thus they can communicate via the Scratchpad, and synchronizing across a threadgroup is fairly cheap. Meanwhile communicating between different threadgroups, or synchronizing between them, can range from expensive to impossible, depending on exactly what you want to do.

There is no natural way to enforce the time order in which threadgroups execute, and different threadgroups may well be sent to different cores.

You may be able to use atomic variables or careful communication through the L2; but, overall, thinking *within* threadgroups is easy and fast; thinking *across* threadgroups is tricky and technical, not least because exactly what is promised in terms of cross-threadgroup ordering and synchronization is very different from one vendor to the next, and even within one vendor's successive designs.

At this point, let's use the above references to see the actual numbers for Apple's GPU design, and where it allocates (or does not allocate) resources.

So the number of *marketing-threads*, ie *lanes* for one core of the Apple GPU is given as 768..3072. In other words, if each thread/warp/wavefront uses the maximum number of registers it can, a GPU core can maintain state for 768 lanes [limited by number of registers]; if each thread uses a minimal number of registers then one core can maintain state for up to 3072 lanes [limited now by the number of other elements of the state, eg the PC, the threadID, the stack pointer and so on, that need to be maintained].

Dividing $768/32=24$, $3072/32=96$, more usefully we see that a core can maintain state for up to 96 threads, with a minimum of 24 threads in the case that each thread is maximally greedy about registers.

This tells us the number of threads the core can maintain, as pending.

My guess is that this translates into something like

- each quadrant can maintain up to 6 active threads (ie threads actively being considered for execution in the next cycle)
- this set of 6 comes from a pool of up to 24 pending threads per quadrant. Ideally at any given time the quadrant
 - + has 24 threads pending (but may have fewer if there just isn't much work left to do, or if each thread is using lots of registers)
 - + of those 24, six are active, and of the others some are waiting on a DRAM load, others could run right away, and will be swapped into active state when one of the six active threads takes a cache miss.

Now think about this in terms of threadgroups. Recall that a threadgroup may consist of say 16 to 32

warps.

We can see that at any time perhaps less than one, up to three, threadgroups will fit on a single core at a time. In the worst case a single threadgroup (of 32 warps) will have to execute “sequentially” across a core, that is at any given time somewhere only 24 of those 32 threads will be executing, they will gradually finish execution and new warps will be started up on the core, until eventually the threadblock is done. This also makes it clear that there may be no time at which *all* the threads of a threadblock are executing simultaneously, which means some sneakiness is required to implement threadgroup barriers and synchronization...

Fortunately the sneakiness is lightweight, so fast, low complexity, low energy.

Returning to our scheduler, we can see that the Scheduler has more degrees of freedom to think about.

Assume a simple one-core GPU. If we are executing one kernel, then essentially our task looks like

- 1- break the kernel (which may consist of millions of lanes of execution) up into threadblock sized units
- 2- send those threadblocks to the Core for execution

3- at any time maintain on the Core as many threadblocks as will fit (so that we have as large a pool as possible of threads to toggle between active and inactive as we are forced to wait for DRAM).

Tasks 1 and 2 are interesting in themselves, and we will get to them. But consider task 3. We are constrained in how many threadblocks we can pack onto a core by how much Scratchpad each threadblock requires. Once that is satisfied, we are constrained in how many threads (ie warps) of each threadblock we can pack onto a core by how many registers each thread requires. We want to be able to pack at least two threadblocks onto a core so that, as we approach the end of a threadblock, we are able to start executing threads from the next threadblock; which requires the Scratchpad storage to be at least twice the maximum Scratchpad any one threadblock can allocate.

If we are executing two or more kernels then the balancing problem becomes trickier.

On the one hand if we try to finish up, as much as practical, one kernel before starting the next, we will probably achieve optimal data reuse (in the L1, L2, even the SLC) between different threadblocks of the kernel.

On the other hand, if we interleave the two kernels, each may be using different hardware (maybe one is doing a lot of FP arithmetic, while the other is mostly using the texture unit) and they can run together without even really slowing each other down.

You could imagine running simulations to try to discover, across a large range of pairs of kernels, the optimal scheduling strategy.

Fancier would be for the compiler to indicate a few flags with each kernel that suggest whether it is optimally run shared or alone.

But best of all (and what Apple pushes hard, unclear if nVidia cares as much) is to take lots of measurements and report them up to the various scheduler levels which can then decide at a fine-grained level (based on things like degree of memory reuse, or degree of using different execution units) whether or not to interleave which successive kernels.

what sizes are we working with?

The Apple GPU allows a thread to use up to 128 4-byte registers, or 256 2-byte registers. This comes to maximum 512B of register storage per thread per lane; or 16kB of register storage per thread. With 24 maximum capacity threads active, this comes to $24 \times 16\text{ kB} = 384\text{ kB}$ of register storage across the whole core. (Or perhaps better thought of as 96kB of register storage per quadrant?) If threads use fewer registers (going all the way down to only $128/4=32$ 4-byte registers) then we can pack more threads into the core, up from 24 all the way to possibly $24 \times 4 = 96$ threads.

You can see that 384kB is a large amount of storage! Larger than the L1D and L1I of an M1 P-core taken together!

This means that register storage is implemented as an SRAM, not a traditional register file; and it means that accessing a register from that register SRAM is about the same degree of latency, and energy overhead as accessing L1 cache. We'd like to avoid that if possible, hence modern GPUs use some form of (more or less sophisticated) *register- or operand-cache*, which we'll discuss in great detail.

If you compare this to nVidia's most recent Hopper design, then nVidia's quadrants hold 64kB of storage (vs Apple's 96kB), and a core can hold a maximum of 64 (vs Apple's 96) pending warps.

Apple give a threadblock up to 32kB of storage, from a total pool of (what looks to be, on the M1) 64kB of Scratchpad storage. This goes along with an 8kB L1D.

nVidia, conversely believes that some algorithms can run much faster if they have more shared storage (think eg histogram or FFT) and their flexible per-core SRAM of 256kB allows for (as of Hopper and CUDA9) up to 227 kB of Scratchpad storage per threadblock, with the unused portion used as L1D. Note that, as mentioned, if a threadblock does choose to use a large enough portion of Scratchpad, then only one threadblock at a time will fit onto a core, with inefficiencies as we approach the end of the threadblock and run out of separate warps to execute simultaneously. Win some, lose some! nVidia now also has a very interesting extension to how Scratchpad is used which we will discuss when we get to the nVidia history section.

implications for program/algorithm design

If we think of a threadblock as say 1024 lanes (32 threads), each “processing” say 8..32 data element, then a threadblock handles 8K..32K elements.

This 8K..32K is a useful value to keep in mind because synchronization (across space and time), and communicating via the L1, within a single core is fairly cheap. Ideally you want to split your task up into data blocks of size 8K..32K (or larger if you can figure out how) so that your threadblocks are as large as possible and, once launched, do as much work as possible.

If we extend our vision to a second dimension of doing work on each core, we have of order 10 GPU cores (8 for M1 up to 78 for M2 Ultra) so the unit of work to think about at the entire GPU level is around 320K data values. Operations of this size *can* be synched across cores, within the GPU, at more cost, and communicating via the L2; however doing this does take some degree of care.

If we have work that extends beyond this ~320K values, depending on the precise details, we may want to split the job (the “grid” of work) into smaller pieces called *passes* that run successively; and we may want to use a different type of algorithm to stitch the passes together. For example we may want to use a radix sort to sort a certain amount of data (let's say it's 100,000 units) within the GPU, to create

sorted blocks each 100,000 elements in size. At this point we may want to use a different type of sort, like a merge sort, to stitch the blocks into a final sorted array.

The point is that rather than submitting to the GPU a single kernel (ie a single sort algorithm) sized at ten million units, consisting of an ungodly number of threadblocks that frequently execute extremely expensive cross-threadblock synchronization operations, the smart programmer usually uses a nested set of algorithms. So maybe we submit to the GPU a first set of kernels that process the blocks by radix sort, with all communication only within threadblocks; then a second set of kernels that process the results of the first pass via merge sort? It may even be the case that we split the task; doing the first set of radix sorts on the GPU, then the second pass of merge sort on the CPU.

It's a mark of the naive GPU programmers to assume every program should be written as a single algorithm executing on a huge grid of data, and that the GPU will just sort this out. (Even though this model is encouraged by intro GPU programming texts.) This is, more or less, like writing your code assuming you have 100GB of DRAM available, and ignoring the costs of virtual memory and cache locality, just write the code to a huge address space and assume it works. In both cases abstracting out performance costs is fine while you are learning your craft, how to program on the GPU or CPU; but in both cases once you start to tackle large problems it's time to put that simple model behind you and program in a way that's aware of VM and caches, or that's aware of the costs of synchronizing within vs across threadblocks.

Ultimately we are trying to get as much value out of locality as possible. The hope is that by moving an instruction (from a threadblock) or a data item into the L1I or the L1D it will then be used repeatedly by a large number of the threads in play on this core. All the numbers involved (number of threads in play, size of threadblocks, size of caches, etc) are all attempts to achieve as much locality as possible without hurting the situations where locality does not exist.

So it's not surprising that all the major vendors have converged on similar sorts of designs and numbers when viewed from a high level.

A second point is that you have to think of "threads" as much more lightweight entities than for a CPU, because they are managed by hardware, not by an OS.

"Creating" and "destroying" GPU lanes is basically free;

Creating and destroying the executable blocks of GPU lanes, ie threadblocks, is very light weight; think a little more expensive than the cost of a function call.

Creating and destroying kernels (ie the actual thread of code that will be repeated over and over across many many cores and lanes) is somewhat expensive; that you should think of somewhat analogous to (though perhaps less than) the creation of an OS thread. This cost, in particular, has varied wildly over different GPUs over the past decade, but has generally become cheaper and cheaper.

I already described our "dot product" program split into 16 threads each handling part of the dot product.

But for a GPU, think splitting this on much more fine grained level, given the costs we gave above.

You'll create one (not completely free) kernel, but the units of work created below the kernel are most not worth worrying in terms of their creation cost. Worry about synchronization, worry about data reuse. But don't worry about the costs of creating threadblocks and threads.

Most of this splitting into finer and finer parts is done by the compiler. You indicate that you want to operate on a 1, 2, or 3D array of a particular size, and the compiler will create an optimally sized pool of identical threads each accessing some sub-array within the array. More or less most of what, in CPU code, would be looping, incrementing pointers and array indices and testing bounds, is handled by the compiler and Thread Manager in terms of how it implements the bounds and run-length [ie how many array elements are operated on] of each thread.

GPU address spaces

How does “storage” (ie “memory”) work?

First think about address spaces.

global (device) address space

We have a global address space (essentially seen by the entire GPU kernel, also seen by the process that created this kernel, and what we in CPU-land think of as the process's address space). Apple, like everyone, calls this *device* address space.

Requests in this address space may go out all the way to DRAM/SLC, and can be cached in the GPU's L2 or L1. This means that this address space provides the potential for sharing (or, if mismanaged, interference...) between any lane of any threadblock of any kernel created by a single process.

However you don't get this sharing for free; GPU's have a weak memory model and so if you require a write in this address space by some lane to be visible to some other lane, you may need an explicitly programmed barrier of some sort. The GPU programming model (ie Metal) may provide some implicit guarantees (like at the end of a threadblock everything in the L1 cache may be flushed to L2), but the details of this are finicky, and seem to depend to some extent on whether you are performing general GPU compute vs executing a particular stage of the graphics pipeline, so I don't understand them much.

The usual situation for a non-Apple GPU is to have a CPU with attached RAM, and a GPU card with attached VRAM or HBM. A few years ago these physically separate pools of storage were mostly also logically separate. The GPU saw an address space that was limited to the card-attached VRAM or HBM, and special DMA calls by the driver were required to move data between this card storage and the CPU DRAM.

Now modern GPUs have a memory management system with MMU, TLB and the rest, and one of the ways this is used is in a form of paging, so that if a page in CPU DRAM is accessed by the GPU, it is transparently copied over into VRAM/HBM, and vice versa. This relieves the programmer burden of manually having to synchronize data between these two storage locations, and allows for code that shares pointers and pointer-based data structures between CPU and GPU. However this is still not as efficient as Apple's scheme, relying as it does on moving entire pages between one pool of storage and the other. Apple's scheme in contrast provides a single pool of storage and access at the line, rather

than the page level; ie lines that are touched by both the CPU and GPU will (conceptually, there are some important details...) move between CPU and GPU caches like they would between the caches if the cores of two different P clusters successively touched the line.

per-lane “address space”

At the other extreme, we have *lane-local* storage. This is the equivalent of TLS (thread-local-storage) as used by CPUs and as accessed by, eg, the GS register on x86; or data stored on the stack.

Currently (as far as I know) all the GPU APIs only support a lane-local *stack*. They do not seem to support the wilder variants of what is possible with the C++ `thread_local` keyword (possibilities like per-lane *globals*, or per-lane *dynamic memory allocation*)...

The *hardware* scheme Apple uses to handle lane-local storage (assuming I understand it correctly...)

- seems like it could handle per-lane `malloc` or *globals*, if at some point there is value in adding this functionality

- ultimately places this storage somewhere in physical address space so that (ultimately, behind the scenes) these values live in L1 cache through L2 through to DRAM.

In principle, I believe, you could use a fairly large per-lane stack if you want, and the data will just move in and out of L1, L2 and SLC as usage dictates with no size limits (except, perhaps, whatever the Metal compiler and runtime enforce).

- I think (but am not sure) that the Apple hardware scheme for implementing this allows for fairly rapid teardown and reuse at the end of a threadblock so that the usual situation is that successive threadblocks are constantly reusing much the same *physical address space* for the stack; and this is usually sitting in L1 and L2; but pathological cases that created a large stack for one particular kernel will, I think, eventually flow the high stack addresses out to SLC and DRAM, probably lasting until the creating process' address space is terminated and the relevant physical page is recycled.

The summary of the above is that, for all vendors, per-lane storage boils down to per-stack storage which operates as you would expect. You write code as usual, which allocates local variables on the stack. As far as possible, these will be translated into per-lane registers, but special cases (eg when you use the address of a local variable as a pointer, or when you allocate an array on the stack) will allocate as storage which, to the developer, behaves like normal storage on the stack, and which is implemented as storage living (ultimately) in the global address space, and cached at L1D, L2, or SLC depending on reuse and such factors.

This all sounds innocuous but has some, uh, interesting consequences...

For example, one obvious way to structure this per-lane storage is if you imagine the baseline stack as, say,

- a *per warp* stack base,
- a sequence of 32bit words,
- then we implement each word's address in physical address space as the per-warp stack base + the word address×32

This has the nice consequence that stack reads and writes are naturally coalesced to make best

use of L1 bandwidth.

But it **also** means that there's no natural way for one lane to transparently access another lane's data. That would be fine, except that the C++ spec says that you can take the address of a variable on one thread's stack, share that address with another thread, and have that other thread access the variable. You can argue that this is stupid language lawyering (and that's the realistic situation right now; it doesn't work and if you want it to work you'll be told not to be stupid!) but it does show the kinds of weird things that can happen given this sort of design if you think you can just port over unmodified CPU programs.

scratchpad storage

In between these two is local storage that is shareable across the lanes or SIMDs of *one threadblock*. This lives in a separate address space that only exists for the duration of the threadblock; and is implemented in a separate physical SRAM.

Everyone uses a different name for this (eg *Local Address Space* or *Scratchpad* or *Threadgroup Address Space*). For the most part I use Scratchpad as this makes it clear that we're dealing with something conceptually different from "normal" (ie stack, or device) memory.

Apple has one pool of Scratchpad per core but it is used in two ways.

Compute kernels use *threadgroup* address space, and can allocate variables in this space. Codewise this looks like a stack allocation, with visibility and lifetime limited to a function; but access-wise it's more like a global variable in that it's visible to all the other lanes in a threadblock, and whatever one lane writes to this variable, other lanes will see (with collisions, race conditions, etc, if you are not careful about ensuring that all lanes of all warps write to different locations...)

The primary characteristics of this storage are

- it's faster than using device address space
- but slower than registers
- it is addressable
- it disappears (ie gets wiped and reused) when a threadblock ends execution

So suppose eg, we are writing a reduction in the simplest possible way:

- We would use registers to reduce within a SIMD (collapse 32 input values to one output value).
- We would use threadgroup storage to communicate and perform the reduction *across different SIMDs*. So each SIMD writes its one reduced value out to a successive address in Scratchpad. Then we barrier (make sure all threadgroups have done this stage one reduction). Then we load the 32 successive values from Scratchpad, again reduce them within the SIMD, and write out the final reduction (result of 32×32 reductions) to device address space. (We can't write it to Scratchpad because that will disappear once we exit this threadgroup).
- We would use device address space (hopefully usually hitting in the L1/L2 cache, and possibly requiring some caution to ensure stores by one threadblock are seen in the appropriate order by other threadblocks) to communicate *across different threadblocks* to perform the entire grid-wide reduction.

I don't believe Apple (or anyone else) provide dynamic allocation (ie something like `malloc()`) for Scratchpad storage, and it's less clear than for the per-lane storage case that they may someday do so; or that there would be much advantage in doing so.

The second great value of threadblock storage, apart from speed, is that, unlike registers, this address space is indexable, ie you can look up by address. Thus it may be of value for tasks like histograms or hashtables.

But Scratchpad storage is not especially large; right now Metal limits you to a total usage of 32kB for one threadblock.

Meanwhile Fragment shaders (ie the end part of the graphics pipeline that texture and render triangles to the screen, in a form known as *Tile* shaders) can use this same physical storage (basically a pool of SRAM) as *imageblock* address space.

In this usage the data has the same sort of constraints and advantages as *threadgroup* address space (it only exists as long as this threadgroup of the fragment shader kernel exists; it cannot communicate to code outside this threadgroup; it's fast, etc) but now the addressing of the storage is two-dimensional, the storage acts like an image *tile*, and hardware takes care of some of the addressing overhead (*x* and *y* dimensions, and also, if you choose, possible multiple channels of data, eg RGB). It appears to be the case, as of M1 and M2, that there is 64kB of SRAM per core, to be split dynamically between threadblock storage (possibly there's more than one threadblock executing, and each may want say 30kB of threadblock storage); and however many Tile shaders and Fragment shaders (ie other threadblocks, but using their threadblock storage as image tiles) can fit.

graphics “buffers”

Beyond this there is a whole world of detail in how data resources and data storage are allocated in different graphics APIs, with constantly varying and changing details regarding the sort of stuff compiler, language, and API designers care about (types, mutability, whether you can have pointers into them, degrees of indirection, etc).

I know nothing about this stuff except that it's the subject of furious discussion between proponents of the different ways Metal, Vulkan, D3D etc handle them; but none of it seems to be relevant to what the hardware does, so I've tried to ignore it as much as possible. Mainly I've tried to understand the GPU as a general purpose throughput computer, and have ignored the graphics-specific details.

The basic summary, for Apple, is that rather than using `malloc()` or `new`, you allocate an MTLHeap of a certain size and then (if you wish) suballocate from within that heap. MTLBuffer, MTLTexture, MTLResource are variations on this theme.

This may seem irritating – why not just use the standard well-known language or OS allocators? – but it's to optimize the storage to specific characteristics of GPUs; by specifying various details of the buffer you can optimize their use, and the driver and/or hardware can track lines or pages of data with differ-

ent properties.

For example you can specify some block of storage as *memoryless*, which sounds like an oxymoron but essentially means the storage is to exist purely as temporary storage (meaning that, for example, it should be retained as close to the GPU as possible and when the storage is deallocated, the various cache lines that were allocated to it can be invalidated without writeback. [You could also, for example, suppress snooping at the L2 level and cache coherence with the CPU, but I don't know if Apple do that.] *Private* mode similarly means the CPU will never see the data (so you could again optimize snooping and coherence) but now the data can't be treated as temporary, it may be re-accessed later by the GPU.

pointer types?

Consider the existence of all this different address spaces. While they route to physically distinct storage, are they actually distinct addresses? Conceptually yes, but what about in reality? Do I actually use different instructions to read from Scratchpad vs Local vs from Global memory? Obviously this can lead to complications if I have to ship, for example, every function in multiple versions that take every combination of pointer types...

nVidia make a big deal of the fact that all their pointers are ultimately identical, so that (more or less) the hardware looks at high bits of a pointer to determine where to route it. This makes so much sense that I would expect (by now, if not during 2010s) everybody did this.

The above refers to pointers as the hardware sees them. A second dimension is pointers as the language sees them. Apple's MSL, for example, qualifies pointers as being of type (among other things) *device, constant, threadgroup* [ie Scratchpad]) which implies some degree of the problem at the source code level, though perhaps Templates simplify things and allow this variant source functions all to collapse to a single assembly function?

Improvements to this basic GPU model

So that's all background. Even within this limited explanation, a few things should stand out as soon as you start thinking carefully:

- the simplest version of the device we have described executes one kernel at a time, and does so by executing one threadblock at a time on each core.

There's no memory protection of any sort.

There's also no "time" protection of any sort, ie no clean way to deal with a kernel that's out of control, beyond an interrupt that resets the entire GPU.

So we'd like to make this design more capable and more performant. In order of complexity, we might consider adding

evolution of GPU: sharing

- packing more than one threadblock onto a core at once
- as one threadblock finishes (so that say 9 of its 32 warps have completed and 23 are left) start the next

threadblock on the available 24th slot on a GPU core, so that there's no dead time while most of the GPU core is unused and we're waiting for the last few warps of a threadblock to complete

- executing more than one kernel at a time, but all from the same process (so there's still a single, non-virtual, non-protected address space). But don't mix kernels on cores, allocate some kernels to some cores, other kernels to other cores.
- allow threadblocks from different kernels onto the same core. This is a big improvement because it has the potential to allow different hardware blocks to be used simultaneously. This especially matters for graphics, since we can now interleaves threads executing vertex processing with threads executing fragment processing.
- allow kernels from different processes. At this point we can no longer delay the issue of memory protection; we need to give our GPU cores an MMU/TLB subsystem.

evolution of GPU: pre-emption

At this stage we still can't pre-empt cores (ie we can't time slice cores; we start threadblocks and essentially wait until they are done). This limits our control if we want to be able to do things like run a large background computation on the GPU while still ensuring that it displays graphics at full speed with no glitches.

The first attempts to do this used the compiler to insert "possible pre-emption points" at certain point in the code (for example possibly after a miss to DRAM, when the thread will be moved to inactive anyway). This is better than nothing, but doesn't give you full control if, eg, the background code enters a long loop of pure calculation never making a memory reference that misses in cache.

So the final step, more or less, is to make each core pre-emptible on any instruction boundary, like a CPU. This does not, however, imply that the GPU will be treated the same way as a CPU. GPU pre-emption is expensive (the context of a thread of execution is large) and is usually not necessary, in that most threadblocks, even most kernels, are short-lived. So, for the most part, a system will try to naturally swap in high-priority and swap out low priority code at natural breakpoints (end of a threadblock, or end of a kernel), only forcing pre-emption if there seems to be no alternative.

It's an interesting point that the above evolution is remarkably recent, which has as a consequence that many people with strong opinions about GPUs are living at some point in the past where some of the above features have not yet been added. This constantly bedevils communication if you're trying to reference one of these modern aspects of a GPU and the person with whom you're communicating is not even aware that this is an issue.

A particularly problematic form of this is the GPU academic literature which, until very recently, is mostly locked into a model of a single kernel executing to completion on a GPU with no sharing. Many ideas that seem reasonable in the context of such a model turn out to work less well when the reality is

that the GPU is being shared, and required to switch from one type of task to another (eg interrupting a long-running computation every 60th of a second with screen updates).

The mirror image of this problem is the game “literature”, ie reviews and internet opinion pieces, where, once again, gamers care about nothing except playing a single game, so while they might care about how well the game’s multiple kernels appear to be shared on the GPU, they still care nothing for leading edge functionality like pre-emption, or even VM/TLB support.

Timeline

Before we get to the details of the Apple GPU, we’ll run through the past 20 years or so of GPU history. This is interesting in its own right, given you an understanding of what the competition has prioritized, but also often illuminates the choices Apple has made.

Prehistoric days (before programmability takes off)

- Direct3D 8 (2000) provides for very limited vertex and fragment shaders, so that’s around the time that GPUs start to be designed as at least vaguely programmable, hence the sort of architecture we are describing.

A year or two later people start talking about the idea of GPGPU with initial languages like BrookGPU. (These initial attempts have to do things like fake the computation as a graphical process, eg as a fragment shader; the hardware did not yet have the idea of a compute shader decoupled from graphics concepts.)

This may seem ridiculous, but we still haven’t got *that* far beyond it. MSL (Metal Shading Language) has shader in the name, and you have put up with some degree of irrelevant graphics nomenclature (and a *lot* of graphics-forced delay) even if what you want to do is write pure throughput compute programs with zero relationship to graphics.

- Direct3D 10 (2007) pushed the idea that the graphics pipeline was generically programmable using a common programming language (HLSL), rather than a fixed function pipeline with minor programmability bolted on in a few places. So now we’re fully in the modern world.

- Direct3D 11 (2009) introduces Compute Shaders as first class entities, so now we’ve supposedly achieved full decoupling of throughput computing from graphics.

- Meanwhile in the hardware world, the first interesting hardware is Tesla (2006) with the first version of CUDA coming out in mid 2007. OpenCL (2009) shows that everyone has become interested.

So by 2007 we have hardware that conforms to the GPU idea we have described. Let’s see how that evolves. As we go through this, remember that we are describing what were at the time leading edge, physically huge chips on separate large cards. Phones, naturally, lagged behind these!

CUDA1/Tesla is really the lowest version of this idea. As best I can tell (it's hard to be quite sure whether some limitations reflect CUDA or Tesla HW)

- it can support only one kernel at a time,
- appears to have no “thread-level” synchronization primitives, only atomic memory operations
- can support up to 32 warps resident on a core, coming from up to 8 threadblocks (compare with M1 supporting up to 96 warps on a core)
- scratchpad memory of 16kB
- a special cache, holding only constants, of 8kB
- a texture cache (essentially invisible to the programmer, and not used by compute)
- small L1I but **no** L1D

You can read about Tesla here: <https://www.anandtech.com/show/2549>

(This article will give you a feel for how little GPUs were understood at the time, 2008, and the obsession with graphics and gamer details.)

nVidia Fermi 2010

We get Fermi (and CUDA3) in 2010. This gives us

- a first set of thread synchronization primitives and memory fences
- up to 16 kernels on the GPU (but they all have to be from the same process). That sounds good but there are some implementation constraints (to be described when we get to Kepler in 2012) that limit how well this works in practice.
- up to 48 warps on a core
- an L1D cache, and more precisely a common pool of 64kB cache, variably split between Scratchpad and standard L1D cache, while allowing the maximum amount of Scratchpad storage per threadblock to increase to 48kB.

Once you have multiple L1D caches along with an L2, the question arises of what sort of coherency is promised. (This does not arise with Scratchpad memory because that's a different address space from the L2/DRAM address space; and is only live and relevant for one threadblock, which only executes on one core.)

The initial nVidia rules are very simple – no promise of any sort of coherency, between L1D and L2, or between any two L1D's.

To make this practical, for the most part the L1D is used purely for lane-local storage (eg the per-lane stack). The developer can indicate that certain loads from device address space should be cached in L1D, but this only makes sense when you know that your use of the loaded data is read-only.

There's probably some behind-the-scenes use of the L1D cache for short term memory coalescing (reads and writes from different lanes and different warps, over a few cycles), but the details of this are not given.

- Something strange with Fermi is that nVidia reduced the maximum number of registers per thread from the 124 of Tesla to 63. It seems like they believed that spilling registers to the stack (which can be stored to the new L1D cache) would now not be much of a problem. This limit of 63 registers was retained through the first version of the successor Kepler architecture, but was later seen as an error, and in fact reversed to 255 registers in the later Kepler designs and subsequent.

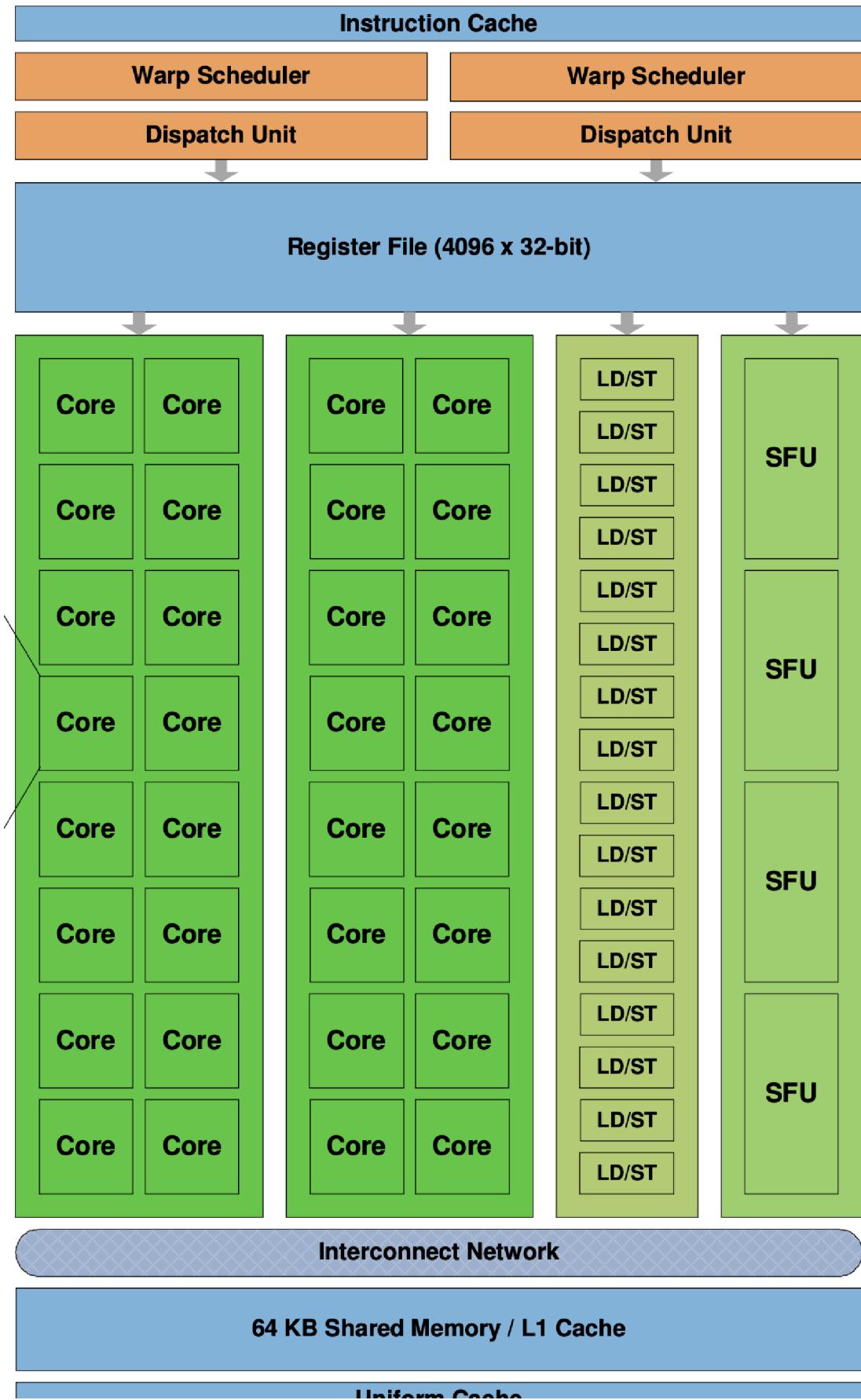
superscalar issue (general GPU discussion)

- In Tesla (and earlier) 32-wide warps were executed on hardware that was 8 lanes wide, by running each instruction through these lanes 4 times. There were various other hacks involving running parts of the chip at half rate relative to other parts.

With Fermi this *mostly* goes away.

As best I understand it, Fermi should be thought of as two schedulers in the front, issuing to six execution units's in the back, each handling a half-SIMD of work. One scheduler handles even numbered warps, one handles odd-numbered warps. The two schedulers hit the L-cache, (presumably alternating, but with one taking over full time if there is an L-cache miss); perform decode, and produce two half-SIMD-wide instructions which are sent to Dispatch, which waits for registers and instruction units to be ready then sends two (half-SIMD) instructions to whichever of the six “execution units” are free.

You might think the diagram below shows four execution columns. In fact there's some artistic licence. and there are six sets of execution units: two columns of int/fp operation, a column of load/store, two different SFU [Special Function Unit] columns (one handles interpolation, one handles other special functions) and a Texture column. Later versions of the design added a third int/fp column, so ideally we're dispatching two half-SIMD instructions to two of seven execution units each cycle, with the understanding that many units will execute their instructions over multiple cycles.



I think the idea behind this is to give the system more flexibility: a long latency instruction will lock up one column of execution units while it executes, but the other columns can keep going. This is one way to handle an issue I will mention repeatedly – each core has multiple different execution units, eg load/store, FP, texture; and in a perfect world each of these would be occupied 100% of every cycle. Obviously this can't be achieved, but weirdness like this nVidia split scheduling goes some way to ensuring that one unit gets given a lump of long-lasting work, while the other units can simultaneously operate on a sequence of faster instructions.

The details of this dispatching change over the next few designs, so it's not worth obsessing over them.

The points to notice are

- multiple schedulers. Eventually this will evolve into a design of 2, 4, or 6 mini-cores, each with a scheduler, but sharing an I-cache, so that if one of the schedulers hits an I-cache miss, the other can keep going.
- various different attempts to achieve superscalar issue.

The simplest way to achieve this (and one that is copied by everyone) is to provide a limited number of the more rarely used execution units (SFU and texture units, and not here, but soon, load/store units). This allows a single instruction to those units to occupy multiple cycles, and while that work goes on, work can be dispatched to the more commonly used (eg FP and integer) units.

Another way to achieve this, that we see here, is to feed instructions from two different warps (which clearly have independent registers) into a single pool of execution units.

Yet another way to achieve this (which nVidia later abandons then returns to) is to make your primary execution units 16 operations wide, but encode everything as 32-wide warps. This means that

- (more or less) in the same amount of silicon we can provide two cores rather than one
- each core operates at essentially half the frequency (takes twice as long to process warps) BUT
- in cycle n I can issue the first half of say an FMAC instruction, then in cycle $n+1$ it's obvious that the second half of that instruction has to occupy that column of execution units, but the scheduler can find a separate instruction to send down a different column of execution units

You could imagine multiple other ways of doing this, and they have all been used by someone!

For example a VLIW design could utilize long (possibly variable length) instruction bundles that specify instructions targeting 2 or 3 or 4 of the different columns of instruction units.

Or two adjacent instructions could have a bit set in one of them saying that it can be paired with the other for dual execution (like Pentium UV execution).

Or you can run the bulk of the GPU (ie all the “real” execution) at a low frequency, like say 800MHz, and the scheduling at 1.6GHz, so that each cycle the scheduler is able to think about and schedule two instructions.

But almost all these ideas from the earlier days of GPUs have been abandoned. Why?

In a sense they solve the wrong problem. They try to solve the problem of maximum throughput per area (keep every unit ticking every cycle) but

- it's not clear that's the right problem to solve. Obviously you want to use your execution units as

much as possible, but with modern designs the real constraint is power dissipation. Multiple frequency schemes that run parts of the GPU extra fast (and hot) are especially problematic.

- it's not clear this is the right way to solve the problem. The biggest costs on a GPU are, like a CPU, memory movement. The details vary but, roughly, moving data from RAM into an execution unit costs hundreds of times the energy cost of an FMA; even just moving data from the register SRAM into the execution unit costs $\sim 10\times$ the cost of the FMA. This suggests that, especially for a vendor like Apple, your highest priority should be minimizing data movement (and getting maximum value out of every data movement that occurs).

This cost of data movement exists at every level, and means that every GPU vendor has adopted some form of “register cache” that holds register values that are expected to be reused soon (to reduce the cost of their data movement from the register SRAM). We’ll discuss Apple’s version of this in great detail. But a consequence of this is that trying to make the scheduler superscalar (via any of these tricks described) is constrained by the bandwidth of the register cache – if the register cache can only supply, say three source operands per cycle, and the most common instructions require three (FMA) or two operands, then the various costs of widening the register cache may make a more superscalar design even less appealing.

There is one *somewhat* superscalar issue that has become common. My primary FPU is a 32bit FPU, and it doesn’t take that much extra logic to allow this to operate as either an FP32 unit or as two FP16 units acting on a SIMD2 vector. This scheme fits easily into the pre-existing data-flows because an FP16 SIMD2 vector fits into a 32-bit register and into the appropriate slot in a register cache. So both AMD and nVidia now support this. AMD in fact (in the most recent CDNA products) take this a step further. These products, unable to compete with nVidia in the AI training space, have tried to find as their niche throughput computing for HPC and STEM, meaning that they prioritize FP64 performance. This means the basic compute unit is not an FP32 unit but an FP64 unit (along with the same for registers, etc); and the instruction set allows for operations on a SIMD2 of FP32 elements.

Apple, prioritizing energy first, has gone in a different direction. Rather than overriding the FP32 unit to allow it to handle a SIMD2 of FP16, Apple in fact provide both FP32 and FP16 execution units, and route to one or the other as appropriate. The additional cost of performing FP16 within the FP32 unit was considered to be sufficiently high that they were willing to spend the area to have two units, each optimized for just one task. This doesn’t answer the question of whether it would make sense to retrofit the FP32 unit to also handle FP16 SIMD2 operations, and maybe one day they will do that? Retrofitting SIMD2 to your design does, of course, also require some compiler changes, so even once you decide to do it, you may still only publicize it a year or two later, once those compiler changes have been added.

Fermi high level scheduling improvements

So compared to Tesla, the low-level scheduler improvements include

- one scheduler can keep generating instructions even if the other suffers an L-cache miss
- instructions are buffered between Scheduler and Execution, so that a long latency instruction (eg one

that has to be executed four times to pass 16 data lanes through the 4 Special Function Unit execution units) doesn't force its Scheduler to halt

- it's unclear how deep is the buffer of instructions between Dispatch and Execute, but one hopes that it can hold at least a few instructions (say six or eight). To the extent that this is provided, and to the extent that Issue is able to look at this pool and maintain only ordering *within* a threadID, not require ordering *across* threadIDs, we can get some re-arrangement relative to Scheduling to take into account which instructions have all required resources (registers and execution unit available) vs which instructions are still waiting for resources.

Something like this seems to be occurring because there is a Scoreboard (ie a constantly updated table stating which execution units are in-use, and which registers are in the process of being calculated but not yet fully calculated).

So there seems to be some degree of "out of order execution" but only out of order in terms of interleaving one thread relative to another.

The point is there is apparently some degree of dynamic decision as to which two instructions execute next, rather than a fixed barrel scheduling.

Beyond the above, with Fermi, the higher level kernel/threadblock scheduling also starts to be taken seriously.

Tesla has one basic scheduler per core which can (under limited circumstances) issue two instructions. [This "dual issue" claim has to do with specific details of how parts of Tesla run at half or a quarter the clock rate of other parts, and is not especially interesting in a technical sense.]

And supporting only a single kernel means there are limited scheduling things to think about.

But once you have multiple kernels, you can have scheduling at multiple levels:

- to what extent do you interleave vs sequentialize kernels; ie which kernel goes next, and across kernels how do you divide resources (number of cores, and how many threadblock to send to each core)?
- on a core, you will have a pool of threadblocks, some of which have been allocated registers and other execution resources, others of which are known as potentially executable, but have not yet been allocated any resources).

Once again as old warps complete, which threadblock do we choose to provide the warps to fills the newly vacant slots?

- how dynamic are these decisions? For example I can split a kernel into threadblocks, allocate a set of blocks to each core, and call it a day. Or I can retain the threadblocks in a pool associated with the high level scheduler, and *dynamically* hand out threadblocks as cores run dry. Obviously this second model does a better job, especially if some threadblocks will complete a lot faster than others.

It's interesting to think about scheduling as an abstract problem, and the various ways you might try to optimize it. However an important question to ask is "how much information am I allowed to use?"

You can do a better job if you have information about the next kernel to be scheduled (for example if you know that it makes heavy use of load/store, maybe schedule it to interleave with a kernel that

makes heavy use of FP?) So maybe the compiler should annotate each kernel with some information of this sort?

But even better is if the core is constantly collecting data of this sort, and reporting it up to the various higher level schedulers...

- finally, from the pool of active warps, if more than one is ready to execute, which one gets the next execution cycle?

Fermi at least starts to also think about this issues.

You can imagine, for example, arguments for clustering like with like (stick with the same warps from the same threadblocks on the same cores) as much as possible for maximal memory/cache reuse.

Alternatively there are arguments for maximal interleaving (independent threads won't all want to use the same resources, or won't all miss to DRAM, at the same time).

Or, as much as possible, scheduling the oldest (kernel, threadblock, warp) as the highest priority...

With Fermi these issues became prominent and, as you might imagine, generated a substantial academic literature. However that literature has, to be honest, mostly been less effective and less interesting than the equivalent in the CPU space, largely because it's constantly lagging the (rapidly changing) GPU designs. Unless you are investigating fairly abstract principles, a simulation based on the details of, say, the Fermi scheduler setup is uninteresting given how different these details are in each of Fermi's successors.

At the highest level Fermi calls its scheduler, deciding which threadblocks from which kernels to send to which cores, the GigaThread Engine; at the lowest level it is described as having a "Dual Warp Scheduler".

limitations in Fermi

It's not all perfect! And (like its successors) when you look closely you see that in many ways Fermi is less of a GPGPU design than it appears. nVidia throughout the next decade is somewhat split, with the highest level designers really caring about expanding GPGPU functionality in multiple directions, but then the lower level implementors unwilling to spend the transistors to enable that full vision.

- Thus, for example, Fermi still has some idea of "graphics mode" vs "compute mode", and switching between these is remarkably expensive.

- There's also what is essentially a hack to handle the different address spaces.

Tesla used different instructions to load from different address spaces, and this made writing vanilla C code, or passing around pointers, somewhat tricky.

Fermi handles this by moving from (multiple) 32-bit address spaces to a single (64-bit) address space, where the high bits indicate which particular address space is being referenced: *local* [ie per-lane stack space], *shared* [ie threadblock space], and *global* [which, remember is RAM on the video card, not an address space easily also visible/sharable with the CPU; there's still no idea of VM/MMU/TLB.]

In 2011 nVidia introduced CUDA4 which doesn't change hardware, but does provides what is called "Uniform Virtual Addressing". This is not what you think it might be – we still don't have an MMU or a TLB.

What it does do is synchronize how the 64-bit addresses are allocated so that "system memory" (ie CPU memory) and various blocks of address space on different GPUs (you could have, say, two Fermi cards plugged into a PC) are all allocated into different regions rather than stomping on each other, which in term means that pointers passed from one GPU to the other will behave as expected.

I *think* the system memory that's made visible corresponds to the virtual addresses of the *one* process that is using the GPU, not to a general ability to read and write whatever you like in DRAM (which even in the more innocent days of 2011 was surely clearly a bad idea!)

All this means that there's at least the basic infrastructure in place for programming in a way that utilizes two GPUs, but I think this is still very much non-transparent – the developer creates two sets of kernels submitted to the two GPUs, can't expect much in terms of a shared L2 and fast synchronization, there's no automatic load-balancing, etc. In other words, this is still a long long way from, say, an M1 Ultra!

You can get a Fermi analysis (slowly evolving to be less game based...) here: <https://www.anandtech.com/show/2849> and a discussion of CUDA4 here: <https://www.anandtech.com/show/4198/nvidia-announces-cuda-40>

nVidia's white paper is here: https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf

nVidia Kepler 2012

Next up is Kepler (2012)

- We now have four (rather than two) schedulers. The obvious guess is that each handles a quarter of the threads present on the core, indexed by the lowest 2 bits of the threadID, but this is never stated.
 - We've again bumped the number of warps visible to a core, up to 64; but remember that at any given time many of those will be "pending", ie placed on a queue waiting for DRAM.
- nVidia doesn't say (neither does Apple or anyone else) but my guess is no more than perhaps 16 or 24 warps (ie 4 to 6 per scheduler) can be actually active and considered every cycle as a candidate for fetch/decode/dispatch.

Each scheduler can dispatch up to *two* sequential instructions from the thread it chose to schedule this cycle.

We have, essentially, as execution units, sixteen columns of execution units. Again the diagram is somewhat misleading, trying to be pretty rather than accurate. The simple execution units (int/fp) are now a full SIMD (32 lanes) wide. So better to think of these paired up ignoring the spit by 3s.

So we have six columns of 32-wide int/fp cores; two columns of (16-wide, so multicycle) LD/ST units; two columns of FP64 units; four columns of 16-wide SFU units (two visible, two the invisible

interpolation units); and two columns of 8-wide texture units

So you can view this as an evolution of Fermi, something like:

- a scheduler generates two instructions per cycle,
- giving us a pool of up to 8 instructions to dispatch;
- these go through an arbiter which issues them to up to 8 of these 16 columns of execution units.

Once again the hope is that even if some of these columns (ie execution units) are busy executing multi-cycle instructions, from the pool of four dispatchers we can keep finding close to 8 instructions to dispatch and issue every cycle.

Another way to look at this is that if you consider the “core” execution units, and particularly the FP units, as the “main” execution units, as many people do when counting the number of FMAs a GPU can execute as a measure of its performance, then we can feed eight instructions per cycle (albeit with constraints, like they have to be Pentium-style UV paired, and not sequentially dependent) into those 6 FP units. In other words, even if we execute 6 FMA’s per cycle, we still have a little extra capacity available as two additional instruction slots, eg for load/store.



- It's still not clear quite how the instruction loading works.

When you have one or even two schedulers, presumably they more or less take turns to access the L1I, and load in say two or four instructions to be held in a local buffer, so overall there's a match of instructions to the dispatch bandwidth.

For Kepler I assume we still have that model, but it's probably getting a bit tight to ensure that we don't occasionally starve one of the schedulers of instructions.

- For Kepler we add Shuffle instructions, which allow data to be moved across lanes within a single SIMD.

- In late stage Kepler we also get a 64-bit *within-lane* funnel shift, ie in each lane, take two 32b registers rA and rB, concatenate them into a 64b register, and extract some contiguous run of 32 bits into register rC.

- Fermi used a Scoreboard to dynamically schedule which from the pool of queued (ie already decoded) instructions to send for execution. Kepler retains this Scoreboard for tracking when destination registers become valid for *non-deterministic* execution units (like load or texture) but for registers generated by deterministic execution (ie math or shuffling) Kepler delegates this work to the compiler. What's done is something like a "wait-count" is appended by the compiler to each instruction saying how long it has to wait relative to the previous instruction. So now rather than Scheduling always having to look at the Scoreboard to see which registers have become available as fully calculated, frequently Scheduling only has to decrement a counter for each queued instruction, and when that counter reaches zero, then we can assume the required registers are available.

This mechanism saves energy (and perhaps some area?) relative to a traditional Scoreboard tracking every instruction.

The paper <https://arxiv.org/pdf/1903.07486.pdf> *Dissecting the Nvidia Turing T4 GPU via Microbenchmarking*, in chapter 2, discusses how this (along with other scheduling) information is encoded in the instructions, though the details have changed repeatedly through all the different architectures from Kepler (2012) to Turing (2018).

- The combined L1D/Scratchpad is bumped to 128K, with a maximum of 112K available as Scratchpad (ie minimum L1D capacity is 16K), but shared over multiple Threadblocks, since a single Threadblock is limited to 48K of Scratchpad.

- The Fermi texture cache was just a (graphics mode) texture cache.

The Kepler texture cache can, in "compute mode" be used as an additional read-only cache. You can think of this as a special purpose storage for table-lookup.

There was a period during which people even exploited texture functionality to perform interpolation on the values within this table, which sounds cool, except that texture interpolation details change from every design to the next, and are performed at minimum quality just good enough for "fool the eye" and no better! So after some rude shocks people concluded using this as anything more than just a cached table was a bad idea!

multiple kernel submission

- Kepler (and CUDA) start to improve the situation of kernel re-ordering.

Like so much in nVidia history, the first stage of supporting multiple kernels was something of a hack.

Suppose I write code that submits kernel A, to be followed by kernel B.

To what extent does kernel B have to wait to execute after kernel A? You could imagine one situation, where the two are completely independent and can execute simultaneously; versus a second situation where I expect B to depend on the results of A, so I can't execute B until A is done. So how do I indicate this to the GPU?

At the CUDA level, I think this can be done by creating independent work queues; the API contract is that work within a queue is serialized, while the work of different queues has no mutual dependencies.

That's fine, but what about the hardware? It appears that Fermi flattened these multiple CUDA queues into a single hardware queue.

This means the only opportunity for kernel overlap was at the queue boundaries (suppose Queue1 has jobs A1 and B1, likewise Queue2, and these are flattened by the hardware).

We now force A1 to execute to completion, then B1 can execute simultaneously with A2, then A2 has to complete before B2 can execute).

Kepler provides multiple hardware queues so now we can execute A1 and A2 together; as soon as A1 finishes we can start B1, and as soon as A2 finishes we can start B2.

Technically this is kinda obvious, but everything needs to be invented/discovered a first time, and nVidia thought this is important enough to give it the brand name *Hyper-Q*.

As usual with these sorts of things, there's fine print. In this case, the fine print includes that Kepler (still) distinguishes between a "graphics" mode and a "compute" mode, and toggling between them is slow. These (up to 32) simultaneous kernels can only co-exist while in compute mode; in graphics mode we're still limited to one kernel.

It's also unclear whether a *single* core can support *more than one kernel* at once. Note how all these pieces are slightly different:

- there's multiple processes (or not),
- multiple kernels (or not) possibly all from the same process [but possibly different kernels are always executed on separate cores], and
- threadblocks from different kernels occupying a single core.

Even at the lowest level, in one cycle do we schedule (possibly multiple) instructions from one threadblock (we will see that this seems to be a PowerVR limitation at this time) or (possibly multiple) instructions from multiple threadblocks.

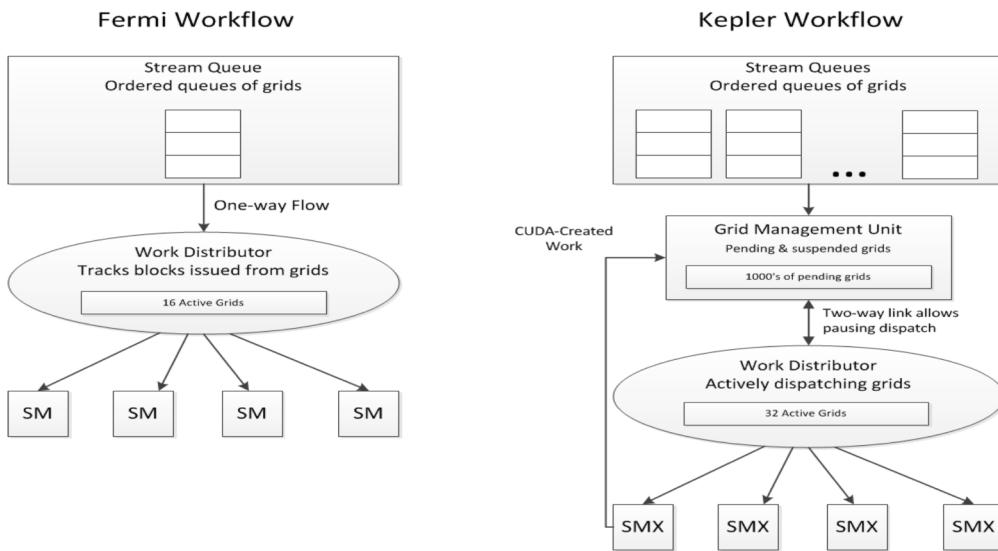
Obviously being able to support and schedule anything anywhere is desirable, but each increase in flexibility also means more of various types of tables stored in different places that have to be referenced every cycle.

An alternative bottleneck to performance is the cost of submitting a kernel to the GPU driver, running the driver on the CPU, and having the kernel transferred to the GPU. Better would be if the GPU were able to launch kernels internally.

For example consider a grid-based PDE solver. After the solver executes a first pass, it might want to refine the grid in some locations while coarsening it in other locations, then running again.

In the past this would require writing out some sort of information to the CPU, which in turn would have to launch a second pass kernel to be transferred to the GPU; now this work of defining the second kernel and launching it can happen within the GPU and without CPU involvement. This is given the brand-name *Dynamic Parallelism*.

This diagram shows something of the difference and how these two technologies are implemented (an SM or SMX is what Apple would call a GPU core).



computational improvements

Kepler is where nVidia really begins to optimize its high end for large scientific usage.

- Atomics performance is brought up to, mostly (when uncontested) the speed of L2 load/store operations
- Support for a wide range of 64b atomics is added (though not yet FP64 atomics)
- Performance of FP64 is dramatically improved
- Mechanisms are now provided for moving data between GPUs (on the same PC) or via RDMA (network to another PC).

At this stage nVidia is only talking about “several” GPU cards collaborating on a large problem, without numbers; they probably have in mind about four cards or so.

(These all remain areas where Apple is still dramatically behind nVidia)

On the graphics side, one of the last pieces of weirdness from the fixed graphics pipeline days is ended.

In those days, textures were treated as special “objects” whose properties (address, size, sampler characteristics, etc) were stored in special “binding” tables set up before a grid launch. This effectively meant a maximum number of textures available to a process, leading to strange hacks like packing multiple textures within a single texture atlas occupying only one binding slot.

This nonsense goes away with Kepler providing “bindless” textures; treated in the way a CPU person would expect, as objects to be passed around via a handle and used/reused as often as one likes. The previous “binding” operation is now handled dynamically and invisibly by hardware moving these texture properties as necessary from RAM into storage in the texture units and cache.

You can get a short analysis of Kepler here: <https://www.realworldtech.com/kepler-brief/>
The nVidia white paper is here: <https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-product-literature/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf>
with https://www.nvidia.com/content/pdf/product-specifications/geforce_gtx_680_whitepaper_final.pdf discussing specifically the graphics/gaming changes to the design.

2013 CUDA updates

Meanwhile on the CUDA front, in 2013 we get CUDA accessible from Python, an announcement whose long term importance took a few years to be appreciated...

The Apple equivalent might be a way to write code in Swift that automatically runs on the GPU, something like marking a function with the keyword `kernel` and everything else just happens behind the scenes to bridge the control and data flow from the CPU part of the program to this GPU call and back again? Such code would not require any synchronization with the graphics pipeline, and could, in principle, be submitted to the GPU with very low latency.

I suspect Apple have this as a long term goal, but don’t want to say anything publicly until the obviation of all the compromises that have so far been required in Metal/Metal Shading Lan-

guage(MSL). Which means the point at which Metal is no longer required to support non-Apple GPUs (ie AMD, and Intel GPUs).

Somewhat relevant in this context is the idea of C++ *Senders*. Senders are an idea scheduled for C++26 (and currently very much a work in progress) with the goal of providing a transparent way to interoperate between the CPU and the GPU. In theory there are already mechanisms that could do this, for `std::for_each()` or the standard parallel library. In practice all these existing technologies were primarily designed for CPUs and assume implicit sync-points or similar overhead that's lightweight on a CPU, but far too high on a GPU. This suggests 2026 as a (not too far away) date at which people will start to expect that they can write a single program in a single (C++) language and have the compiler generate something that makes somewhat valuable use of an available GPU. Which suggests that Apple should be targeting at least the same sort of timeframe for the same sort of goal in Swift.

Later in 2013 we get the successor to Unified Addressing, namely Unified Memory.

Unified Addressing provided a single address space for CPU and GPU memory, but still required *manual* copies between the two. Unified Memory removes the *manual* aspect of those copies. It can't create hardware that doesn't exist, so copies still occur. But most resources that are allocated by a CUDA program will now be moved around "automatically" by the compiler and/or CUDA runtime. You'd also expect that for this to really work well (ie to be as user-transparent and performant as developers might hope) you'd need some hardware support, and that's the case. This is laying the groundwork for Pascal (2016), which provides MMU/TLB support, and it's obvious how if you have TLB support on both the CPU and GPU you can, in principle, move memory from one side to another at the page level, and only as they are referenced. Certainly not as efficient or elegant as coherence between two caches, but good enough for the 2010s.

Interestingly, to judge from early marketing slides, MMU/TLB support was supposed to be part of Maxwell (2014), but it seems they could not get this to work acceptably, and so that functionality was dropped from the product launch. The later marketing slides changed the Maxwell "one line summary" from *Unified Virtual Memory* to the less glamorous *DX12 Support*.

nVidia Maxwell 2014

In the background, Maxwell represents nVidia for the first time now taking power seriously. Going forward, every design change starts with question about how much power it will require. However we won't track nVidia's power details any further than that.

Overall almost everything new in Maxwell seems relevant to graphics, not compute. I'm guessing that this is because VM support was supposed to be in at least high-end Maxwell, and without it other high-end features could not be implemented.

core split into quadrants

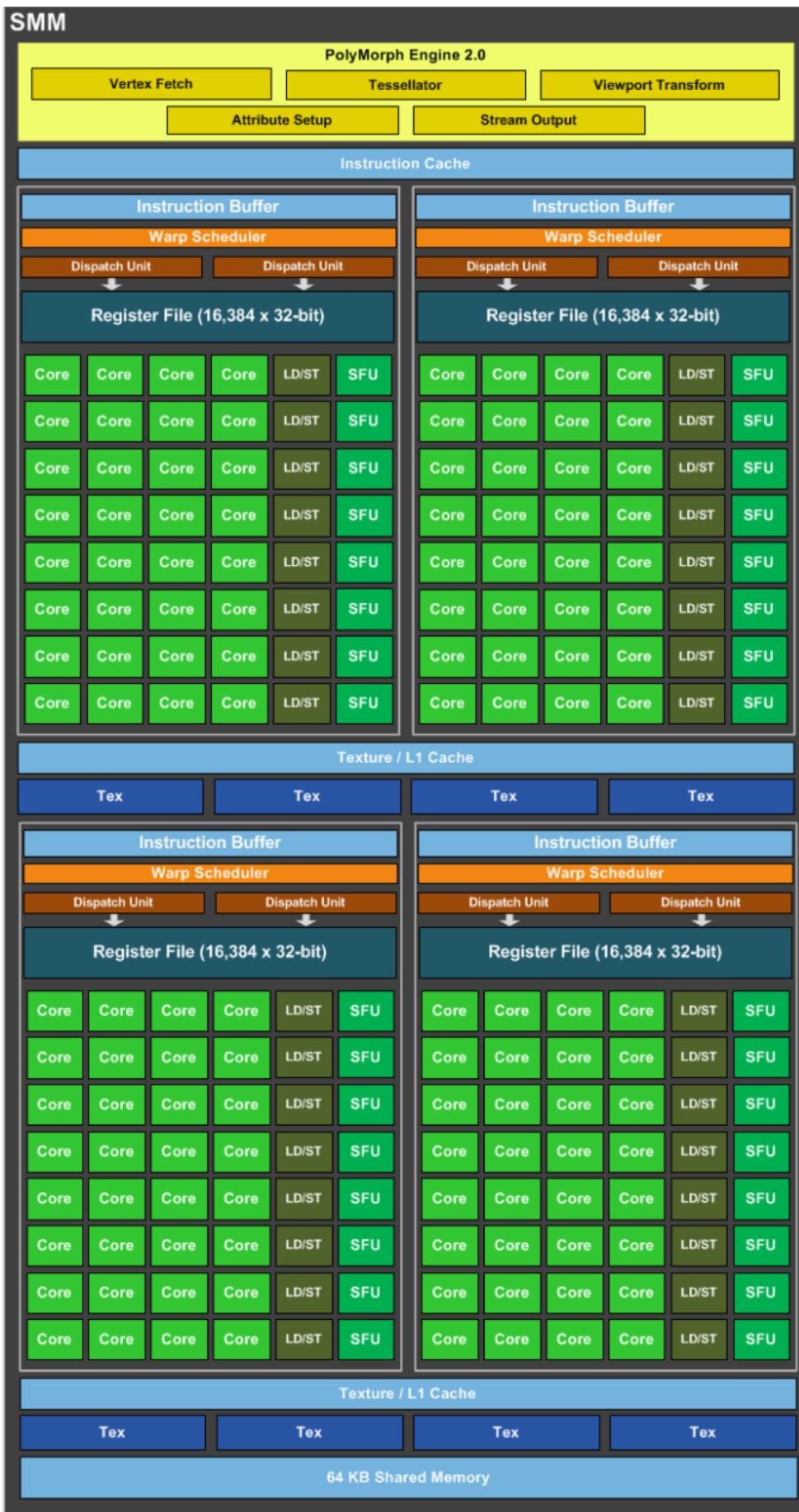
The Maxwell design looks like below (again remember that the pictures are somewhat "artistic"):

Recall that the SMX core was essentially a large flat design that supported (among other things) 6

SIMDs of int/FP execution.

In contrast the Maxwell SMM core is split into four somewhat independent quadrants, with each quadrant controlling one SIMD of int/FP execution. We're arrived at the world where, more or less, one quadrant behaves like a mini-core.

A quadrant has a single scheduler (as always, presumably consulting a list of perhaps 4..6 active warps to decide which Instruction stream to fetch from next), and that scheduler, as before can send one (or two UV-paired) instructions to the Dispatch unit where, as before, there is a small pool of instructions, and some combination of compiler-based scheduling and Scoreboarding decide which instruction(s) get sent to the available execution units, which are now one full-sized (32-wide) int/FP32 SIMD, a quarter sized LD/ST unit, and so on).



From one point of view this is still much the same as before, in Kepler. Overall we still have four schedulers, each dispatching one or two instructions per cycle, to be executed on one or two of a pool of execution units. (As a technical issue, there appears to a single texture unit, and a single FP64 unit, shared by all quadrants, as opposed to four smaller units attached to each quadrant. So there does need to be some degree of cross-quadrant co-ordination to utilize these.)

Localizing the set of decisions from across a full SMX to smaller decisions for each SMM means smaller Scoreboarding hardware, smaller queue hardware, and so on. It also means that for smaller problems, or as a kernel approaches completion, you can start powering down the core in stages, one quadrant at a time.

Note that the register file is split in four, so warps that become inactive because of a miss to RAM have to reactivate *on the same quadrant*.

Overall there is some degree of “performance” efficiency loss compared to Kepler, in that there will be more circumstances where one of the quadrants is overflowed in some way while another quadrant is idle, and there’s no way to move warps from one to the other. But that’s presumably made up for, and more, by energy efficiency, which is now becoming the ultimate limiter of performance (not least because it determines the frequency at which your GPU can run).

We’ve mentioned before three levels of scheduling, one of which is deciding which threadblock to send to which core. Now there’s going to be an additional decision as to which warp(s) from which threadblock(s) to send to each quadrant.

From the set of threadblocks associated with a core, we could associate one threadblock with each quadrant which is very simple; but that may not do a good job of simultaneously using all the different execution units?

Alternatively try to finish one threadblock as fast as possible (giving maximum data reuse in the various caches) by giving it all four quadrants? There are obviously interesting questions here. It would be nice if someone explored the nV patents to see how their solutions have evolved...

The Maxwell Tuning guides has the following interesting, clarifying, quotes:
“...all SMM core functional units are assigned to a particular scheduler, with no shared units. Along with the selection of a power-of-two number of CUDA Cores per SM, which simplifies scheduling and reduces stall cycles, this partitioning of SM computational resources in SMM is a major component of the streamlined efficiency of SMM.”

The power-of-two number of CUDA Cores per partition simplifies scheduling, as each of SMM’s warp schedulers issue to a dedicated set of CUDA Cores equal to the warp width. Each warp scheduler still has the *flexibility to dual-issue (such as issuing a math operation to a CUDA Core in the same cycle as a memory operation to a load/store unit)*”

These suggest that the intermediate level scheduling may be as simple as, for a given threadblock, spreading warps across the four cores based on the two lowest bits of the warp ID.

Another small difference is that the per-core caches have changed. We give up the combined

L1D/Scratchpad cache for a standalone Scratchpad storage (of 96K) while the L1 cache now shares storage with the Texture cache (and the Constant cache also lives here?)

It's still the case that global (ie non-stack) data is confined to the L2 unless you use special compiler flags; likewise it's still the case that local storage (ie the Texture/L1D cache) is used behind the scenes for short-term coalescing of memory requests.

On the technical side, we now get atomics that work in CPU memory not just GPU memory. You can use these to synchronize code that's split between a part running on the CPU and a part running on the GPU.

graphics features

The “Polymorph Engine” stuff at the top is graphics specific. Versions of it are present in earlier GPUs, but I managed to find diagrams that omit it, because we're mostly not interested in graphics evolution.

Just to give a feel for the state of the art at this time (2014) other graphics features mentioned by Maxwell include

- better Tesselation
- better texture compression
- support for large sparse textures (paged textures, where pages are loaded into the GPU on demand)
- a bunch of technical stuff to better support shadows and indirect lighting (brand name VXGI - Voxel Accelerated Global Illumination)
- similarly technical stuff to better support voxel based imaging (including 3D, possibly sparse, textures); part of a D3D 11.3 feature called Conservative Raster
- some technical details to support variants of anti-aliasing (essentially fine control over where the points at which MSAA [Multi Sample Anti Aliasing] is executed)

- ordered shading. The idea here is that suppose Shader A is defined to run before Shader B, but they both ultimately touch the same pixel.

One place you might use this is in diagrams that allow you to see into a complex object, like an engine, where the innermost parts are drawn solid and as parts move outward they are drawn as more and more transparent.

Naively this requires that we run the entire kernel of Shader A, and then the entire kernel of Shader B; but this limits how much parallelism we can achieve.

An alternative is to provide a lightweight “lock” for each pixel so that we run all the warps in maximally parallel order, but require that the “lock” be first acquired by Shader A before it can be acquired by Shader B. (I don't know how nVidia do it at the HW level, but that's the essential idea.)

Now individual warps that overlap the same pixel may be delayed relative to each other, but most warps don't overlap most other warps and so most can spend most of their time proceed at full speed rather than waiting for the previous Shader.

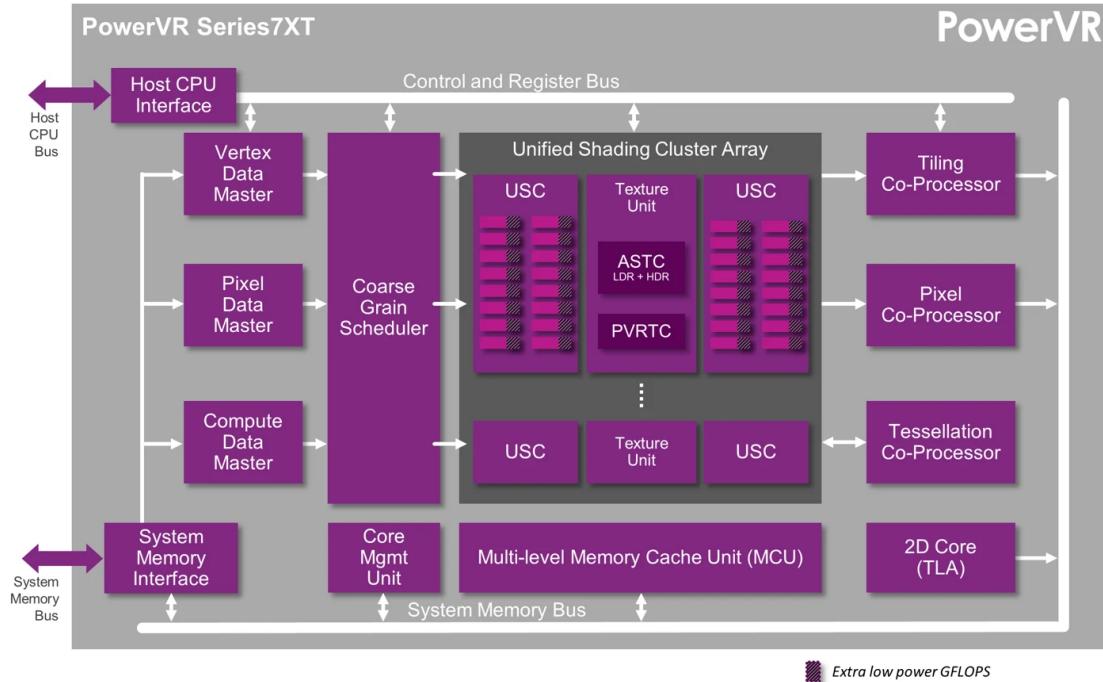
Apple/PowerVR 2014

At this stage, let's temporarily switch directions and take a quick look at where Apple is. We're still in 2014, so recall that the completely Apple GPU (on the A11) is still three years away.

At the time, the obsession of most people was in the details of the graphics systems and the great TBDR vs IMR graphics rendering debate.

This is an issue that doesn't interest me much, but if it interests you you can read about it here: <http://blog.imaginationtech.com/a-look-at-the-powervr-graphics-architecture-tile-based-rendering/>

We can get an overview from this diagram, unfortunately with much less detail than nVidia:



The USC (Unified Shader Core) is essentially the equivalent of an nVidia SMM or an Apple GPU core.

We see various specialized hardware (Tiling, Pixel, and Tessellation Co-Processors) crowded around the USC, basically the equivalent of nVidia's Polymorph Engine stuff.

We see the same sort of idea of sharing texture capacity with USC (ie FP compute) capacity, though the details differ.

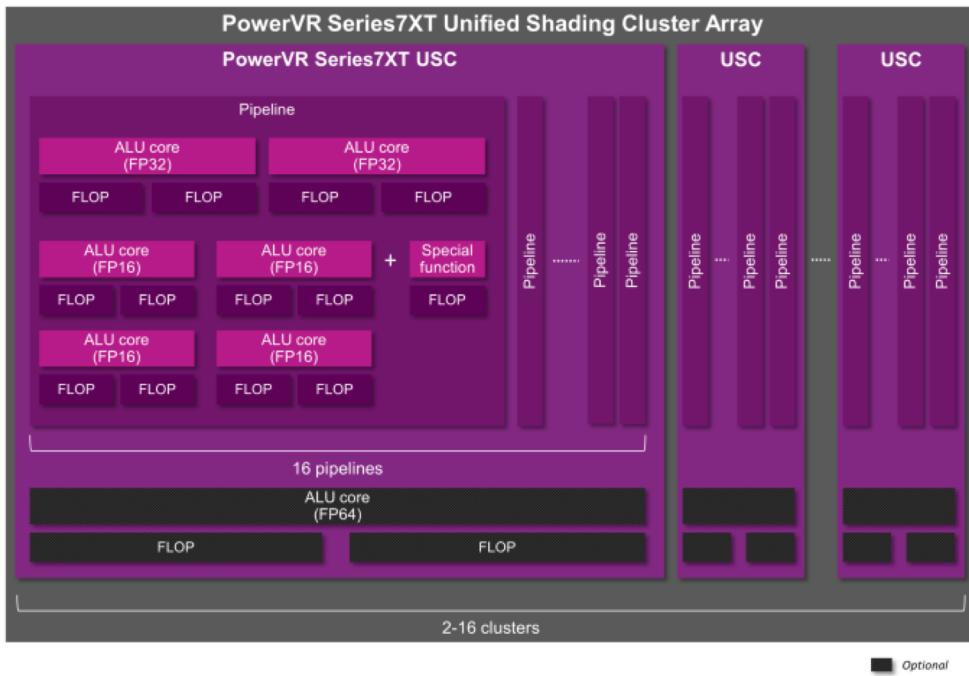
The Coarse Grain Scheduler is essentially the equivalent of nVidia's GigaThread Engine, deciding which threadblocks from which kernels to send to which USC.

The Apple patents frequently refer to the Vertex/Pixel/Compute Data Masters, so it's worth reading the article to at least get a very rough feel for what they are supposed to mean. Vertex and Pixel

appear to be essentially control hardware for the specialized graphics hardware. You'd think their importance would diminish over time, but they are still present in recent patents. As far as I can tell, it's best to think of them as something like high-level (pre-)schedulers that construct, from a graphics scene description, the equivalent of a kernel (or perhaps a threadblock), which is then handed onto to the "real" scheduler.

The main one I really care about is the Compute Data Master, and it seems its job is, similarly, to "set up" compute kernels, meaning something like pulling the instructions in from memory, adding kernels to the Coarse Grain Scheduler's list, maybe defining how to split a kernel into threadblocks, that sort of stuff.

Next lets look inside a USC



What Imagination calls a Pipeline is what nVidia (kinda) at the time called a Core (an nVidia Core was a scalar integer unit and a scalar FP32 unit, not a "full GPU core", which nVidia at the time called an SM). These 16 pipelines are then run at double frequency relative to scheduler, so in one scheduling cycle they execute twice, and so execute a 32-wide operation. We saw that this sort of weirdness with double-pumped hardware was used by nVidia in Tesla, but then removed as the design evolved; it's one way to save area.

So from one point of view this looks like a USC is a half-wide SIMD, like Fermi was (kinda sorta) half-wide; and Kepler jumped to being six SIMD's wide. But somehow it's possible for multiple instructions to be simultaneously dispatched to each USC?

Imagination are very clear that your first guesses as to how this works are wrong! The USC is not executing say 4-wide vector instructions on each pipeline. Neither is it somehow executing two or four "adjacent" lanes on one pipeline.

The way this appears to work is that the instruction set is VLIW (like AMD's was before they switched to GCN). Later, apparently, IMG will learn, like AMD, that VLIW is not that good a solution to the problem; given that you have an easy pool of ILP available it makes far more sense to stick with scalar instructions, and generate ILP by scheduling instructions from multiple different warps each cycle. (ie conceptually split the contents of one "pipeline" into two or four separately scheduled scalar pipelines).

To make things worse, the VLIW design is not especially sophisticated; although you would ideally want to be able to issue any combination of instructions to anything from one to all seven execution units, in practice you're severely constrained as to the combinations allowed. (One of the constraints, for example, though not the only one, is how many source registers need to be collected before execution starts; similarly for how many results will be generated.)

Compare this with the nVidia scheme (which only tries to generate one "real" instruction per cycle, but uses timing tricks [32-wide warp, but 16-wide FP units and 8-wide load/store units] so that frequently two or even three of these different units are executing at the same time; or the clause-based scheme which, via distributed instruction sequencing, allows different types of units to operate somewhat on their own schedule, but which comes with rather complex tracking of hazards within and between clauses.

This all acts as a good reminder that GPUs are nothing but a nested hierarchy of shared hardware! And sometimes what is shared is not obvious. In particular between instruction scheduling and execution, we need both

- somewhere to collect source register values, and
- wiring to route those values to an execution unit.

A "clustered" execution unit like IMG's somewhat reduces the complexity of this wiring; something even nVidia has to take note of (this is the reason that, at this time, nVidia shares its FP32 and integer execution in the same unit, so that they can share datapaths).

Along with this being a VLIW instruction set, some of the sub-instructions targeting FP16 are SIMD2 instructions, for the same reasons that this is done by AMD and nVidia; it's just so easy to fit into the rest of the design.

The other thing we see here (continued by Apple) is using separate FP16 and FP32 hardware, rather than modifying the FP32 hardware to also handle FP16; as explained this saves energy by allowing each of the FP16 and FP32 to be fully optimized for one task, and apparently makes enough difference to justify the extra area. (nVidia, making different tradeoffs of area vs energy, on Maxwell perform FP16 in the FP32 unit, and in Pascal, Maxwell's successor, perform FP16 SIMD2 in the FP32 unit. In principle you

could push this idea further, to provide vec4 support for 8bit math. A few years later AMD would do this as part of their neural support. I've seen conflicting claims as to whether nVidia does or does not do this same thing in Pascal.)

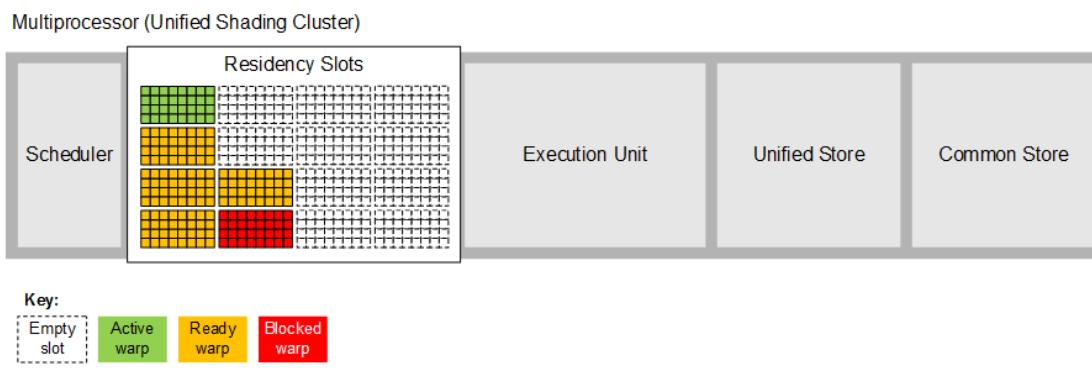
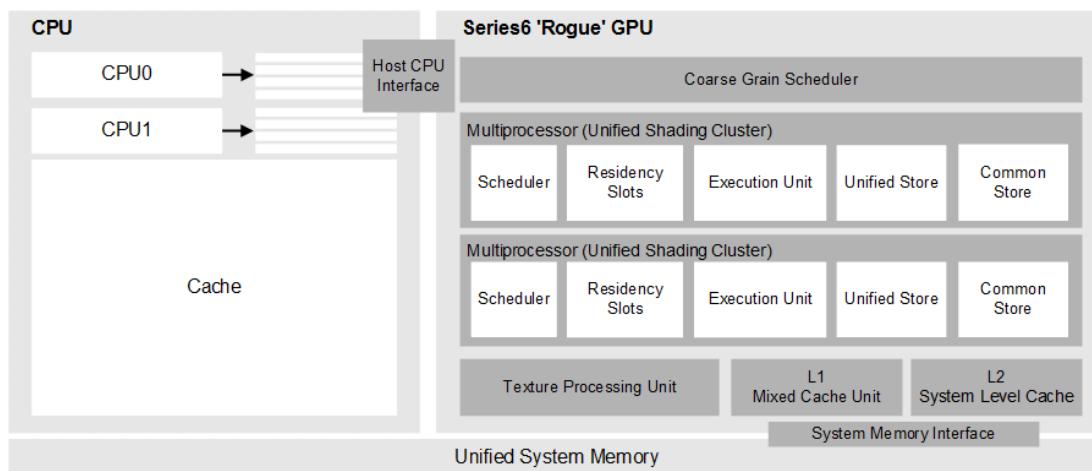
You could go the other way, and have the fundamental unit of control not the 32bit wide lane but, say, a 128-bit wide lane that is treated as a vec4 of FP32's. This was done in some of the earlier GPU ISA's but lost favor as compute programmability became more important. (You can simulate 4-wide vectors on a 32-bit wide execution unit, using the full hardware; but if you want to execute scalar code on 128-bit wide hardware, you are wasting $\frac{3}{4}$ of your execution hardware...)

The sweet spot today seems to be to optimize for scalar FP32, and then reuse that width in the data collectors and data path as best you can for FP16 or even INT8. AMD's HPC oriented designs then use this idea, but flipped to optimize for FP64, with 2×FP32 being bolted on as an easy way to get maximal value from the data collectors and data path, when possible.

(As a bizarre marketing issue, nVidia in fact dramatically throttled the issue of FP16 instructions on their consumer cards! So rather than FP16 being faster than on Maxwell, it was a lot slower. Apparently FP16 had moved from being a sad little format only used by desperate mobile GPUs to a sexy new format used by AI, and nVidia wanted to limit serious AI usage to the expensive cards... Later this silliness of this stance was understood, and the policy reversed.)

Documentation seems to suggest that, overall, at this point IMG is about as sophisticated as Tesla, meaning, for example, only one kernel executes at a time, and with rather expensive setup for each kernel; but with a simpler ISA than Tesla.

The pictures showing the entire “system” look a little more familiar.



Each block (of 32 lanes) represents a warp and pretty much matches what we would expect. So (to the extent that the diagram is trustworthy) we're showing something like up to 16 possible warps present in a core at any one time, some number of which are blocked (waiting on RAM), some are active (being

considered for execution each cycle) while the remainder can be swapped to active as soon as one of the active warps blocks.

The Unified Store (as far as I can tell) is where per-lane data (ie stack data) is held.

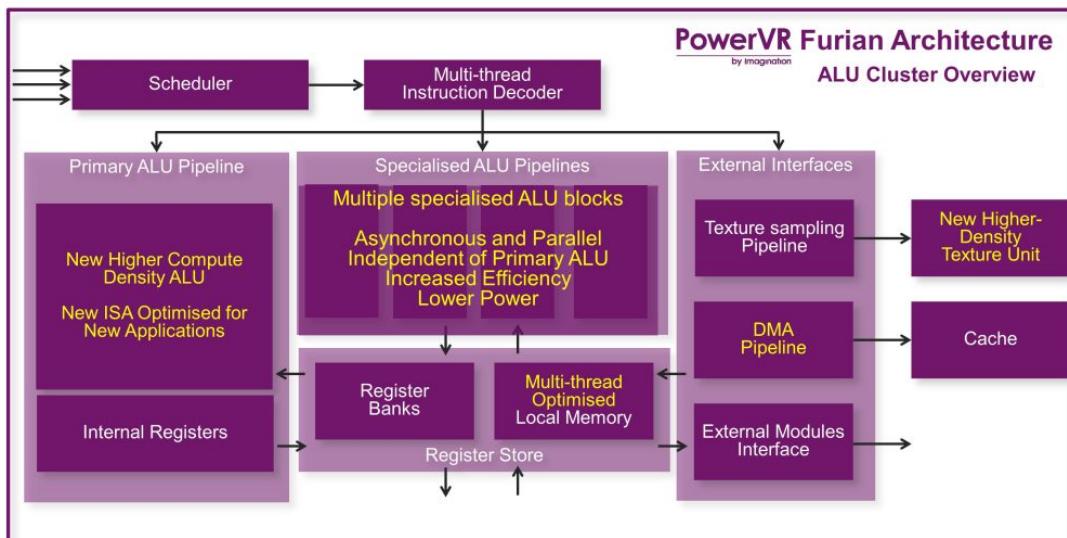
The Common Store is the Scratchpad.

The caches seem to operate slightly behind nV in sophistication – we seem to have a single (for-all USCs) L1 and L2 cache, which doesn't give us the performance of per-L1 caches, but avoids the problems of cache coherency.

On the other hand, ahead of nVidia, there is already MMU support allowing for what mobile GPU vendors call “virtualization”, which in this context means something like each app gets what it considers its own private GPU without being able to interfere with other code, or see other data, executing on that GPU. Even if multi-kernel support is not yet present (something that's not clear) the MMU means caches don't need to be flushed on process switches. Along with virtualization we also have priorities, so that (more or less...), higher priority kernels can “pre-empt” lower priority kernels. Be aware, however, that this is not true pre-emption, it's pre-emption at a fairly coarse granularity: basically when all the *currently executing* threadblocks and tiles are completed, not at some random point in the middle of execution of each threadblock. Still, it's a first step.

PowerVR 2016

Two or so years later this evolves to Furian:



which is more Kepler like:

- 32-wide pipelines,
- pipeline are more like a column of single-purpose execution units,
- it's unclear whether the new ISA is still VLIW or now essentially a scalar ISA,
- multiple "Schedulers" in the sense of finding instructions from multiple different warps to execute in the same cycle.
- an instruction change, which I haven't seen in other GPUs, is the replacement of the two MAC units (or two×2 FP16 MAC units) with one MAC unit and one MUL unit, apparently based on the ratio of MACs to MULs in the code Imagination profiled.

Another thing the Furian documentation states, which seems obvious except that I haven't seen another GPU claim it clearly, is:

consider the situation where multiple threads in a threadgroup want to access different (non-contiguous) Scratchpad addresses; this might happen, for example, with an FFT.

The easiest solution is your Scratchpad only supports one address per cycle, and so this takes up to 32 cycles for one load.

Alternatively your Scratchpad can support some number (how many?) of different addresses and, assuming no bank conflicts, support all those scattered loads in one cycle. The diagram give by Furian suggests they support at least 8 "different" (ie non-contiguous) load addresses per cycle.

How much of this goodness propagates out to loads that can't coalesce, from L1, or L2, or even DRAM? That's unclear – it is notable that the Furian diagram only refers to Scratchpad storage.

PowerVR 2019

Later with their (2019) series A architecture, Imagination very definitely drop anything and everything related to VLIW and converge on a set of simple single-purpose execution units, like what we've seen nVidia use in all their designs.

But with the choice of a very wide warp, 128 lanes rather than 32! You can look at this as a crazy choice; or you can ask yourself why 32 is the optimal SIMD width? Is there a good reason?

Designs have been all over the place, from as low as 4 to at least 64, so it's not utterly bizarre that Imagination might consider this optimal for their particular targets (which are probably ever more heavily weighted to graphics rather than generic compute). AMD in fact currently, on their graphics-targeted designs, allows a warp to be either 32 or 64 wide, and says that 64-wide is generally better for graphics. (They do the usual thing of executing a 64-wide warp on 32-wide hardware by running it through the hardware twice.)

Although Apple is on a different track from Imagination, the two maintain *some* similarities, presumably reflecting similar priorities.

IMG make a big deal about, among other things,

- prioritization of kernels, and the ability to mix realtime graphics work with ongoing background

compute

- dedicated hardware to perform resource allocation, eg registers or Scratchpad storage
- some degree of prefetching data and instructions for upcoming, but not yet active, kernels. (Instructions are easy; data may only be relevant for “structured” data where the GPU knows how it will be used, like geometry lists, or textures?)
- better virtualization in the form of what Imagination call *HyperLane Technology*. I interpret this to mean that essentially every barrier to mixing and matching kernels has been removed; the GPU actively attempts to achieve maximum performance by scheduling different graphics tasks (geometry, compute, data movement, pixel shading, etc) simultaneously, likewise scheduling across kernels from different processes.

This seems obvious to us, but getting there means

- getting rid of all “modes” in the GPU, for example if you have some switch that toggles an SRAM to switch between being used as a texture cache and being used as Scratchpad storage; and
- making sure that you provide the correct isolation (eg through use of an MMU and different address spaces) to ensure that different kernels can’t interact. For example (just one among many) your customers will demand that another process cannot, somehow, sneak a peak at DRM’d video and thereby extract a video stream that is not supposed to leave the device; and
- robust enough pre-emption and priorities to ensure that graphics is never glitched as a result of long-running computation.

- many non-sexy but helpful background improvements like
- + logging various data
- + improved debuggability (include step by step debugging, or viewing registers)
- + performance monitoring, including, eg, seeing which tiles in a frame took an above average amount of time in a particular graphics stage (ie a *heatmap*)

- the presence of a small CPU attached to the GPU and executing firmware, thus avoiding the overhead of tasks that are traditionally executed by the driver running on the main CPU. There is more here than meets the eye.

Consider how the OS actually controls the GPU. The traditional model of an OS is that it runs on the CPU, has access (via obvious memory instructions) to memory space, and communicates with IO via the read and write of magic constants to magic addresses (which map to registers on a USB controller or whatever).

Traditionally GPUs have had to be forced into this model, as a weird kind of IO, where the driver does something like construct a sequence of commands in memory, then write a pointer to that instruction buffer to some magic address, or similar. This is of course a hassle, especially if you are trying to debug! And gives you zero access to the tools available to traditional programmers.

But on an Apple style design (or perhaps even the IMG type of design), there is an alternative! The OS can treat the GPU controller chip (an Apple designed small ARM64 core) as just another CPU, to be

controlled and scheduled like any other CPU, using the same interrupts and assembly code and living in the same address space. At some point this controller hands over GPU instructions to the GPU cores, and at that boundary we again face debugging difficulties, but the boundary from OS to GPU, and the high-level functionality of the GPU (things like queuing up a sequence of submitted kernels, scheduling these kernels, splitting a kernel into threadblocks and scheduling those threadblocks to cores) can be debugged (and thus monitored and optimized) substantially more easily.

It's unclear the extent to which Apple takes full advantage of this different design space, as I have described it, but there's certainly *potential* there for much faster iteration and improvement of this high level GPU behavior, along with lower overhead submission/transmission of tasks from a CPU to the GPU. One could even imagine a future (with the right APIs) where this interaction is essentially as lightweight as two CPU threads co-operating in some way – a one-time OS interaction to spawn the second thread, but then after that mutual interaction based on common memory addresses, IPI's, locks, and atomic variables...

On the graphics side, improvements include

- some details of how the texture cache is used, to make it more efficient (ie cache textures after they have been “processed” in various ways [eg YUV conversion, or gamma correction], rather than repeatedly processing them as they leave the cache)
- better anisotropic anti-aliasing
- blending as hardware associated with tile memory rather than the earlier alternative which was to use a dedicated software blend shader (flexible, sure, but more energy). I'm not sure how Apple do this, but presumably they likewise use dedicated hardware somewhere (and at the tile level is an obvious choice).

PowerVR 2020+

Finally as of very late 2020 IMG announced their series B design.

The most interesting aspect of this is that it envisage (to some extent) a design like the M1 Ultra, namely a distributed GPU. A master GPU handles interaction with the driver and creating work (ie pools of threadgroups from kernels), and secondary GPUs, possibly on other chiplets, request work from these pools. IMG envisage this scheme scaling up to one master and three secondary GPUs. (IMG stress that this “pull” design does not match the multi-GPU schemes of nVidia and AMD, which are based on a “push” design and require coding to the multiple GPU's; the IMG scheme is developer transparent.)

However the most recent nVidia and AMD designs, like the Ultra design, seem based on this same model.)

The latest PowerVR designs are the series C and (as of 2023) D designs which are refinements of the A/B architecture.

Ultimately IMG details matter mainly insofar as they show

- different ways of looking at the problem compared to nVidia
- how IMG (and mobile) was at the time lagging a few years behind nVidia in terms of *largescale* capabili-

ties (L1 cache, multiple kernels, ...).

Of course mobile was ahead in terms of energy saving details.

Each had to flip emphasis at around this time (2014) to match the other.
 - this was where Apple started, so their design (especially the early patents) builds on these ideas.

You can see something of current PowerVR in <https://resources.imaginationtech.com/hubfs/gated-files/gpu/powervr-dxt-whitepaper-en.pdf> (describing the series D design) and <https://resources.imaginationtech.com/hubfs/gated-files/raytracing/rtls-whitepaper-en-jan23.pdf> describing their ray tracing design. I would recommend downloading both of these, but reading them *after* you have read below about the Apple design. At that point you'll be able to recognize many similarities, and you'll be reminded of multiple small Apple features that you see repeated in the IMG design.

an aside: Intel 2012

If you can't get enough of this stuff, then I'd recommend reading: (2012) <https://www.realworldtech.com/ivy-bridge-gpu/>

This gives some insight into the the Intel GPU at the time, and gives a third contrast to nVidia (unlimited power budget) vs Imagination (mobile, so much more limited power and area).

You'll see how everyone has converged on a very similar instruction flow design (the Intel Slice looks similar, for example, to an SMM); it's mainly the precise numbers (how wide to make execution units, how many execution units to place in a core) that change.

However memory access is still subject to many variations with nVidia constantly rearranging their caches, which in turn differ substantially from Intel or Imagination, and with mobile (unlike nVidia) already requiring an MMU.

If you want even more, you can get a feel of where AMD is at right now with:

<https://www.amd.com/system/files/documents/amd-cdna2-white-paper.pdf> and

<https://www.amd.com/system/files/documents/rdna-whitepaper.pdf> and

<https://chipsandcheese.com/2022/09/18/hot-chips-34-amds-instinct-mi200-architecture/>

nVidia Pascal 2016

At this point in our story, we return to nVidia, with Pascal (2016).

In terms of the things we're focussing on, the big change with Pascal is that it takes the Maxwell design (SMM consists of four quadrants) and halves it, so an SM consists of two halves), but you get twice as many SMs. So what? Well think about it:

- From the point of view of *execution* hardware, two Pascal SMs equals 128 "executions lanes" which is the same as one Maxwell SMM.
- But each Pascal SM now has twice the Scratchpad storage and twice the registers (and probably more L1I cache, though nVidia don't talk about this).

So essentially the thought process seems to be that often an SMM cannot fully utilize its four quadrants because the number of registers and amount of Scratchpad storage limit how many warps can be packed onto an SMM, so let's reduce those limitations.

Think of Pascal not as *halving* the size of an SM but rather as *doubling* the number of registers and Scratchpad storage available to four quadrants! Possibly even more important than these storage capacity doublings is a bandwidth doubling to the L1 cache and the Shared Memory (ie what used to be called Scratchpad Memory).

Followed then by some hardware rearrangement to limit how many resources [registers etc] each core has to keep track of.



The instruction flow has become more clear in that nVidia is now explicit that

- every cycle the Warp Scheduler (choosing which one of around 8 or so active threads to service)
 - selects up to two sequential instructions, usually coming from one of the 8 or so instruction buffers (which are occasionally filled from the L1I cache) to place in a queue;
 - and from the queued instructions up to two are issued to an SM half.

Meaning we *hope* for essentially two-instruction/cycle 32-wide throughput. We saw in David Kanter's article on Intel's Ivy Bridge GPU architecture that Intel Gen7, with less flexibility than nVidia, can achieve dual issue about 70% of the time. So we should be able to do pretty well – if memory access doesn't kill us.

Essentially what we hope is that every cycle we are executing a 32-wide FP32 operation, and load/stores essentially come along for free. Note, however, things are not *quite* as great as they might appear because NV's particular ISA details mean you need to perform a fair number of integer operations.

tions (especially multiplies) to generate multi-dimensional array indexes, and those use up FP32 timeslots; likewise for irritating details like handling edges and checking for boundary overruns, if arrays are not a multiple of 32 in width.

Beyond the above, nVidia now starts to take FP16 seriously, and with Pascal can achieve an FP16 throughput double that of FP32. This is done by what nVidia calls a “single paired-operation” instruction, which seems to mean the same obvious FP16 SIMD2 as everyone else.

multi-kernel support

Pascal introduces what nVidia calls *Dynamic Load Balancing*. We still have Graphics Mode vs Compute Mode, with expensive toggling between them; however in Graphics Mode we can execute 1 Graphics kernel together with up to 31 simultaneous Compute kernels; in Compute mode it's 32 simultaneous Compute kernels.

Maxwell supposedly supported this 1+31 workload, executed on a *static* split of the cores into a subset running compute and a subset running graphics. But this static split never worked very well and so this model was not encouraged.

Pascal starts with a similar static split, but excess work at the end of a frame can move from one side of the split to the other as necessary.

Obviously this sort of better use of hardware is always desirable, but it can't perform magic! Under the best possible use cases, this Pascal flexibility in scheduling compute along with graphics seems to buy you about 5..10% improvement relative to Maxwell.

But there are always limitations and fine print!

The default situation is still that shared kernels *come from a single process*; multiple processes share the GPU via successive time-sharing.

Starting with Kepler, the high end nVidia parts have provided functionality called *Multi-Process Service* which allowed kernels from multiple processes to execute on the same GPU, but this is, even as of Pascal, still something of a hack performed by the driver on the CPU, packing these kernels into what looks like a single process, and without either serious security/isolation, or serious QoS guarantees. (This in spite of Pascal providing initial MMU support!)

Which means that its functionality restricted to and only appropriate for specific use cases.

graphics

With Pascal, nVidia starts to *substantially* differentiate the graphics cards from the compute cards, and for the most part I will describe the compute designs. As an example difference, the Pascal graphics design retains four quadrants per SM rather than two halves, with the resultant overall reduced storage/bandwidth capacity per threadblock.

Among the graphics improvements we have the another round of better texture compression (so that effective bandwidth is increased by about 20% relative to 2014 Maxwell, and about 45% relative to the no compression of 2012 Kepler). This is talked about sometimes as Lossless Memory Compression,

sometimes as Texture Compression. The term Lossless Memory Compression sounds like it's generic and might benefit large and "smooth", but non-texture memory resources, like a lot of GPGPU compute, but I think that's just misleading marketing, and that it's standard Texture-only compression.

This sort of texture compression soon becomes standard for everyone. The next step is to also introduce lossy compression (usually rare, and visually lossless) which allows every block to be compressed, not just some fraction of them. For example in 2018 (recall Pascal is 2016) Imagination uses a combination of lossy and lossless compression to reduce texture bandwidth by about 55%, ie giving more than 2x effective bandwidth, as opposed to the around 1.45x effective bandwidth of Pascal. Subsequent to this people will start to also introduce (lossless) geometry compression.

There are also some graphics technologies that were added in late stage Maxwell designs, and slightly improved in Pascal.

Support for *Simultaneous Multi-Projection*. The idea here is that Geometry stages (Tesselation and assembling large scenes out of many elements) are performed once, then the resultant geometry can be used multiple times by multiple shaders. The obvious use case is projecting the same scene twice onto left and right eyes. One can imagine other uses, like displaying the same scene with different shaders designed to pick out different features, for serious scientific visualization, but I don't know if the system is flexible enough to allow different shaders to be applied to the replicas of the geometry stream.

There are a few other technical tweaks to support VR glasses use cases, or even to research new types of displays that can show scene from multiple angles. Perhaps the most interesting (given that Apple Glasses will soon be with us) is a feature called *Asynchronous Warp* by nVidia and *Time Warp* by Oculus. The idea is that *your head* can move a significant amount in say 33ms, whereas *a scene changes* less in that time. How can we use these facts?

We generate an image (actually two!) of the scene that is somewhat spilled over the edges of the glasses, and represents the state of the world as of 20ms ago. But, just before we display these new frames, we warp them (eg slightly rotate and scale them) to incorporate however the head has moved in the 20ms or so between when we started generating the image and right now when we want to display it. So we'll see a scene that's slightly late in time, but is correctly placed in space, and for most purposes that's good enough to feel like no lag.

high-end scalability

Somewhat behind the scenes, but important for nVidia's future growth, are three big changes:

- HBM for larger, higher bandwidth memory
- NVLink for highspeed connections between multiple (eg 8) GPUs all working on a single problem.

To put this in context, this initial version gives a GP100 160GB/s of communication with other GP100's.

Currently, as of 2023, the Grace-Hopper connection provides 900GB/s between a Grace CPU and a Hopper GPU;

and the M1/M2 UltraFusion link provides 2.5TB/s (which is astonishing when you remember that the DRAM bandwidth to one of the Max chips is “only” 400GB/s, so most of that is going to GPU↔GPU communication).

Remember that something like NVLink is not just about bandwidth; it’s also about

- + addressing (easy load/store access by pointer to the RAM on other cards),
- + atomics (again when directed to the pool of common RAM, these work correctly across all cards connected together), and
- + transparent routing (if card A is not directly connected to card C, NVLink will transparently route via card B).

Thus it’s more a “networked memory bus” than just an interconnect like ethernet.

- more “OS-like” features
 - + an MMU/TLB system; and
 - + real preemption, at instruction boundaries, not just threadblock or draw call boundaries).

The MMU allows for user-transparent data movement between GPU memory and system DRAM (but, as I’ve already said, not free data movement; there is still the cost of a page fault, then a data copy over PCIe). This also allows for “virtual memory” for the GPU so that the GPU HBM2 storage can be exceeded, but, as always with VM, while this is fine in small amounts, you don’t want to get into a situation of GPU page thrashing. In principle it could allow for secure simultaneous execution of kernels from more than one process, but it seems like CUDA was never updated to support that.

Meanwhile real preemption helps not just in the obvious ways, but also for debuggability. But switching a GPU mid-instruction is not cheap! For Pascal it costs around 100 μ s aka ~150,000 GPU clock cycles. I’ve no idea quite where this time goes! You’d definitely prefer to switch at a coarse boundary like kernel, or threadblocks, if feasible.

- “psychologically” this is the point where nVidia goes all-in on deep learning. No longer just a side issue, they refer to deep learning everywhere; and much of Pascal seems devoted to making training work better. At this point we don’t yet have LLMs, or generative models; all the excitement is in vision and recognizing items in pictures.

In terms of scaling, the high end Pascal has ~60 SM’s (so the equivalent of about 30 of Apple’s cores, so we’re now at around the high-end M1 Max or low-end M2 Max in terms of “approximate” similarity. The M1 Max (high end), running at essentially the same clock rate, achieves about 1.5x the GP100 score on Geekbench6, (OpenCL for nVidia, Metal for M1 Max); and slightly under Pascal for M1 Max running OpenCL.

Presumably part of this difference is unavoidable OpenCL overhead (which nV works to reduce, and which Apple no longer cares about); and

part is probably a more sophisticated memory system on Apple, all the way from GPU caches to DRAM.

An interesting point is that however Apple is scheduling instructions (Apple does not seem to attempt

any sort of two wide super-scalar or SIMD2 execution) they do OK relative to the way Pascal does this same low-level scheduling.

Another way of making the comparison is that GP100 (the highest, compute-targeted, Pascal device) is rated by nVidia as having 3584 FP32 units and capable of achieving 10.6 FP32 TFlops.

M1 Max is rated as having 4096 FP32 units and (slightly slower clock speed) achieving FP32 10.4TFlops.

Of course Pascal is way ahead on FP64 Flops!

Less sexy, but also important is Pascal, even at this point, has a richer set of 64b atomics than Apple's still very limited set (nothing in M1, in the M2 *only* 64bit min and max).

Additionally nVidia has a scheme, imperfect as it may be, for distributing a problem over, say, 8 GPUs, as opposed to Apple right now being limited to 2 GPUs. (Of course Apple's 2 GPUs are more tightly integrated, with much faster communication, so... win some, lose some.)

In theory GP100 has higher memory bandwidth (4 HBM2 stacks at 180 GB/s per stack; whereas M1 Max has 400 GB/s, but in practice this doesn't seem to matter much. It may be that the smarts in Apple's caching system mean that Apple usually doesn't need to hit DRAM as frequently for the same sized problem? Also Apple is using more sophisticated memory compression (both geometry and lossless+lossy texture compression) as compared to Pascal's only lossless texture compression.

So by this (very hand-waving!) metric, Apple 2021 is at about nVidia 2016 (but of course at much lower power, worst M1 GPU power seems to be about 55W, relative to GP100's 300W). [55W package power, so including RAM and whatever CPU is required.]

To be fair the high end gaming Pascal card, the GTX 1080, while at 8.9FP32 TFlops (so about 15% below M1 Max), but with minimal FP64 support, only hits 180W. So while process, and perhaps some clever design tricks, help Apple reduce power, we're only talking a factor of 3 or so; nice but limited.

On the other hand, what Apple also achieve substantially better than nVidia is reduced power when the GPU is not in use, or used lightly ("idle" power), which is nice for saving phone battery or not heating up a room.

There's another interesting issue for Apple here. Pascal is where nVidia really start to split the HPC and Gaming/Consumer lines. Differences now include not just the obvious, that HPC gets serious FP64, and gaming gets token FP64; but HPC gets "halves" per core whereas gaming gets quadrants (ie HPC gets double the Register and Scratchpad storage per "simultaneously executing collection of thread-groups").

Apple could adopt the same strategy, making the Max GPUs more targeted at compute, the Pro and lower GPUs more targeted at gaming. Or they could conclude that the advantages of one design for testing, layout, and as an API/compiler/optimization target are more valuable than this sort of optimization?

An alternative way to handle dynamic range might be to provide a few low power E-GPUs, with fewer resources, and useful mainly for basic non-3D screen updates? Or alternatively some GPU's are

for graphics (come with texture and tile support and so on; otherwise are just for compute and strip out the graphics hardware), and M1/Pro/Max get different combinations of these options?

You might think the idea of a separate E-GPU sounds dumb, but there's actually kinda a precedent! Look at this very early Apple GPU patent (2012) <https://patents.google.com/patent/US9035956B1> *Graphics power control with efficient power usage during stop.*

The details don't matter much and are clearly way obsolete, but the patent envisages the GPU having two types of GPU cores, one type being "low-latency" the other type high latency. The idea is that once we have moved past an initial chunk of hard work (eg at the start of the frame) we power down the high latency cores (high latency means something like "takes longer to power down") and transfer the remaining small amounts of work to the low latency core(s) which we then also power down as soon as that last work is done.

It's unclear what they mean by "high latency", why some cores might be this way; perhaps something like maybe it takes longer for them to transfer any accumulated state to GPU L2 because they are further away from the L2? Regardless, the big idea I see as interesting in this patent is the suggestion that one core doesn't fit all sizes, and differentiating them (mildly or substantially, though, like E-cores and P-cores, not in ISA/API-visible way) may have some value

Yet another way to handle the issue is to rethink the Pro/Max/Ultra design. Right now Apple creates two sets of masks, for the Pro and the Max, and "reuses" the Max masks to create the Ultra. Imagine instead that one set of masks defined the Pro, like right now; and a second set defined a large graphics chiplet holding nothing but GPU, SLC, and memory controllers. Apple could then (in principle anyway) mix and match systems from a single Pro to two Pros (people who mainly want CPU compute) to a Pro and graphics chiplet ("Gamer machine") to two Pros and a graphics chiplet ("computer artist" machine) to two Pro's and two graphics chiplets ("Ultimate" machine).

nVidia Volta 2017

In 2017 we see the next step in the on-going separation between a compute-optimized core and a graphics-optimized core: we now have two separate architectures, Volta for compute and Turing for graphics.

Once again the SM (still effectively half an Apple core) has changed substantially!



Numerically, we get about 40% more cores and so about 50% improved FP32 FMAC performance. But in fact the changes run far deeper than that.

- Latencies improve yet again. An FP32 MAC took 9 cycles on Kepler, 6 cycles on Maxwell/Pascal, and takes 4 cycles on Volta.

- From two halves, we're back to four quadrants in a core, but each quadrant only holds 16 FP32 execution units. Presumably a 32-wide operation executes over two cycles. But the load/store width remains at 8; so effectively we have doubled load/store performance relative to compute performance, solving the fact that memory relative to compute was becoming dangerously low in Pascal.

Likewise we get more cache and Scratchpad.

We have some idea of the numerical details of Volta, which give us at least a feel for other nVidia designs:

- The L0 I-caches (one per quadrant) are 12kB in size.

- Instructions are 128b (16B) in size, so the L0 only holds 768 instructions. Instructions are large because they incorporate a lot of control data (ie the control data added by the compiler does most of the hazard avoidance that on a different design might be handled by a scoreboard).

- Instructions also have 4 bits of register cache control info.

The nVidia register cache is much simpler than the Apple register cache. There are four cache "lines" each holding four 8B values (one 64b register), each of the four values associated with a particular argument (1st, 2nd, 3rd, 4th argument) of an instruction.

The primary control feature is a 4bit mask in each instruction that specifies that the source register of this instruction should be cached for reuse. There is no caching of result registers. Observers generally categorize this as more about avoiding bank conflicts when reading source registers than about either reduced latency or reduced energy.

- Warp scheduling across quadrants remains very simple. For a given threadblock that has been chosen for execution on this core, warps are assigned to quadrants based on the lowest two bits of the warp.

Consequences (in principle, unclear how much these matter) are that small threadblocks (less than four warps) will always go to the same quadrants, not be spread around; and that there's no dynamic rebalancing of later warp scheduling if the warps on one of the quadrants execute slower than on the other quadrants.

- There's a per core 2kB L1K constant cache, and combined 128kB L1I + constants cache (a natural pairing in that they are both read only)

- All the cores then share an 6MB L2 holding data, instructions, and constants.

- We're back to a shared pool of L1D, Scratchpad, and texture storage. But the L1D is now writeback, not writethrough, so it also allows for fast data exchange between threads of a threadblock. NVidia phrases this as saying that

- + the L1D was optimized for streaming and memory coalescing while

- + Scratchpad was optimized for latency, and

+ this new combined storage is optimized for both.

Making the L1D writeback raises the issue of coherency, but I can't find anything as to exactly what's promised. My guess would be that, in the absence of coding an explicit barrier, there is something like a flush from L1 to L2 at the completion of each threadgroup?

- We're no longer attempting explicit ILP (ie two instructions issued each cycle from a given thread). As I described earlier, we still effectively, for many purposes, two dispatch slots (ie each 32-wide operation starts its pipelined execution over two cycles, so that if the next instruction targets a different execution unit, that unit could start execution).

How is Shuffling across a 32-wide warp supported? I have no idea! Some non-visible temporary storage?

- We've split off integer execution from FP32/FP16 execution, so that integer overhead does not get in the way of FP32/FP16 execution. Or to put it differently, imagine I do something like dispatch FP32 instructions on every even cycle, then I have the odd cycles free to dispatch integer or load/store or other "overhead" instructions without ever reducing my FP32 throughput.

NVidia say that there are about 36 integer instructions for each 100 FP instructions, so this goes a fair way to improving overall usage of FP32 hardware.

- All this means that for a large matrix multiply, FP32 efficiency runs at about 97% of peak possible, compared to about 90% for Pascal.

(This is FP32 matrix multiply running on the FP32 cores; we'll discuss the tensor cores below.)

The scheduling changes, and the execution unit rebalancing, together mean that on Volta we are able to hide almost completely the overhead of load/store, and address calculations.

- overall (for a range of HPC code, but ignoring AI) we get ~60% improvement for ~40% more "cores"

- The headline change in Volta is the addition of Tensor units.

In this first version, they are only capable of a 4×4 FP16 matrix multiply, accumulating into either an FP16 or FP32 destination.

Even so, they accelerate FP16 matrix multiply by 4x over what would be possible using just the FP32 (ie $2 \times \text{FP16}$) units.

This gives us ~100TFlops for FP16. Compare this with an M2/Pro/Max which provides this at about 7TFlops via the ANE, and about 14 TFlops for the high-end M2 Max GPU. (That's for both FP16 and FP32; M1 and M2 appear to have no performance advantage of FP32 vs FP16, nor a way to achieve double FP16 performance by sending half the operations down the FP32 pipeline.)

Tensor cores are pretty impressive if you have a problem that matches them!

Across a wide range of benchmarks (ignoring stuff that clearly benefits from the tensor core!) improvements range from almost nothing to a little over 2x.

The almost nothing cases refers to something that will be a constant issue going forward: memory bandwidth, both to local VRAM, and even worse if we have to go to system RAM.

But it can't be denied that for many workloads, this rebalancing/rethinking of the ALU and cache

© NVIDIA Corporation 2018. All rights reserved. NVIDIA, the NVIDIA logo, Volta, and all other NVIDIA brand names, product names, and/or trademarks are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and/or other countries.

new thread execution model

Probably the most interesting conceptual update in Volta is the handling of control divergence.

Up till this point (and still, for most GPUs) the execution model has been one of a kind of fancy SIMD, with a single thread PC controlling the execution of 32 lanes, and predication masks (mostly handled by the compiler) faking the execution of `if/then/else` branches.

This model has done well in terms of maximizing throughput for a limited number of transistors, but does provide some (not always obvious) constraints as your algorithms become more complex. A particular problem arises when you want interaction of some sort across lanes, but within an `if/then/else` block. This may be exchanging data (eg via shuffle), or may be synchronizing.

For example the `if()` instructions may want to wait, on the assumption that the `else()` instructions will provide the waited for event, but the `else()` instructions *cannot execute* until the `if()` block is fully completed...

Issues like this particularly plague any attempt to provide abstractions, like libraries and complex functions, where the function has no clue as to whether it will be called by all or only some lanes.

The new Volta model can be thought of no longer a single PC for all 32 lanes, but

- 32 PC's, one for each lane, along with
- strenuous attempts to converge execution wherever possible.

When execution is converged (which is all non-branching execution, as in the previous GPU model) things behave as before.

But when execution diverges, we no longer execute all of one branch, then all of the other, as is more-or-less required by the predicate mask model.

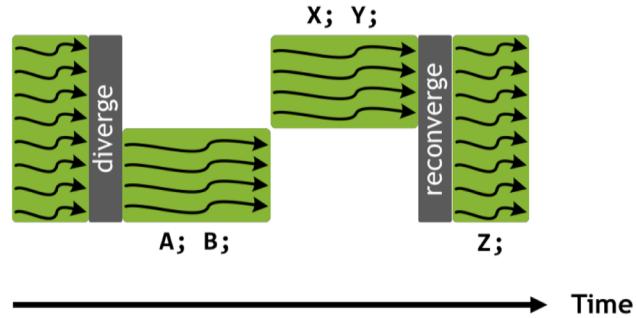
Rather we now have “two threads” with different PCs, whose execution we interleave for optimal performance. nVidia provides the following diagrams to clarify the point:

Old model:

```

if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;

```

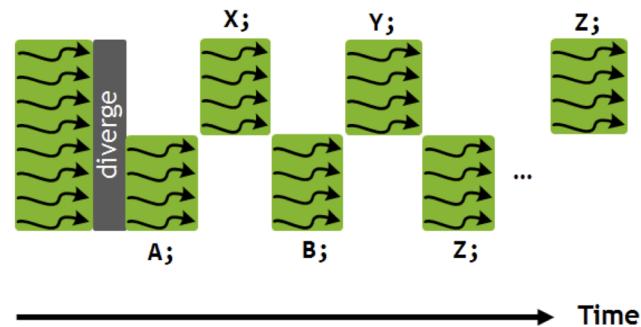


New model:

```

if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;

```



In the old model, if say the threads wished to perform a sync at the end of A, dependent on the execution of X, the code would deadlock; with the new model, performing a sync at the end of A, within a branch, is now possible.

On the other hand, if the new model (for whatever reason) fails to allow for reconvergence at the execution of Z (primarily for technical reasons, if there has been attempted synchronization in either of the two branches) then we will lose performance. The circumstances under which this might happen are rare, and the programmer can force a resync after `if/then/else` if the system does not believe it to be safe, but the programmer knows it is.

This opens up CUDA to a whole new world of algorithms that cannot safely execute on old-model GPUs, and was the next step in CUDA (somewhat justifiably) becoming the standard API for throughput computing. With this functionality, you can start to imagine things like massively parallel graph algorithms where each lane potentially locks or unlocks linked lists (occasionally waiting for a lock held by another lane) without worrying that the wrong lock acquisition pattern will deadlock the GPU.

Along with this new model, a new model of thread co-operation is provided.

The traditional synchronization model allows for

- rapid synchronization within a warp (ie across 32 lanes)
- fairly rapid synchronization within a threadblock (so within up to about 32 warps, the maximum size of a threadblock)
- slow (and tricky) synchronization across threadblocks. Slow because you have to use some sort of atomic primitive not designed for the purpose, and constantly going to memory; tricky because there are no guarantees as to the order in which threadblocks are executed, and it's very easy to deadlock (even if you get lucky and operate successfully on one GPU, you may not be as lucky on another).

CUDA9, in conjunction with Volta, allows for both

- smaller levels of granularity (synching a set of threads smaller than a threadblock, possibly even smaller than a warp) and
- larger granularity (multiple threadblocks within a kernel, or across multiple kernels, even executing on different GPUs).

NVidia calls this *Cooperative Groups*. Essentially you define a (or multiple) *groups* of threads, and then issue sync commands at a per-group level. Very nice!

A different way of looking at this is that: previously certain constraints (like memory layout) forced upon you a certain pattern of synchronizations that were possible. Now you can lay out your lanes/threads as you like (probably optimal for memory access) but define an orthogonal layout of how you want synchronization operations to occur.

If we are being honest, CUDA definitely shows warts here as a consequence of how things have been added over the years. While the functionality described here does exist and is conceptually beautiful, it's messier than it should be to actually use, with somewhat different calls required once you grow your problem from one GPU to multiple GPUs. Maybe one day these language aspects of CUDA will be cleaned up?

The astute reader will have noticed that the Volta diagram replaces the previous per quadrant (or half) "Instruction Buffer" with an L0 instruction cache. I assume that, apart from whatever other flexibility and power saving this change provides, it also helps with the fact that different lanes within a warp may now be executing different instructions from different points in the instruction stream, and an L0 cache is an easier way to ensure that instructions are available as we bounce forwards and backwards along the instruction stream.

high-end scalability

NVLink gets updated. The individual links are faster, and each GV100 chip gets more of them, so NVLink bandwidth off a chip rises from 160GB/s to 300GB/s. It also becomes more of a "general memory"

fabric, not just a GPU memory fabric, so various functionalities like coherency, atomics, and some page table operations, now interoperate with the CPU (at least if the CPU is designed to support the feature...), not just between the GPUs.

Some missing MMU functionality is filled in (for example the Pascal bulk DMA functionality required that all accessed pages be pinned in memory, it could not generate page faults; Volta DMA will generate page faults as appropriate, removing the need for pinning).

Multi-Process Service, the technology for simultaneous (as opposed to sequential) execution of kernels from multiple processes, becomes somewhat closer to real GPU virtualization.

- Much of the work to support this, previously done by the driver on the CPU side, is now done on the GPU;
- the separate kernels are given separate address spaces and fault isolation and
- (some, not much...) QoS guarantees, for example limiting what fraction of GPU resources a particular process can consume under these shared conditions.

In this same area of “OS-like support” there’s better performance monitoring, especially to track page accesses; to allow for better page migration between GPUs and to/from system memory. (For example, if a GPU accesses a new page and wants to copy it in, which page should be the victim? You could just choose a random page, but with counter data you can make an informed choice.)

I keep pushing this issue of multiple kernel support because it seems likely to be ever more important as GPUs evolve to handle every more throughput computing. This multiple kernel support has non-obvious consequences that are important for future design.

For example we all know that bandwidth is an important constraint on GPU performance. When a GPU is running a single kernel, it probably makes sense to optimize the memory data transport and controller for throughput. But once you are running multiple kernels, they may have very different response to memory demands.

Some kernels may have “critical” code, ie large amounts of subsequent code (including perhaps many other warps or even many other threadblocks) is waiting for that code to complete, whereas other kernels may have few such dependencies.

Under such circumstances, it makes sense to start tracking measures of criticality (eg the extent to which a GPU core is halted waiting on memory) and using those, along with kernel priority/QoS settings, to treat some memory requests differently from others.

Once you think about this, this is an issue with many dimensions, but some ideas and data are discussed in (2016) https://people.inf.ethz.ch/omutlu/pub/CLAMS-core-criticality-in-GPUs_sigmetric-s16.pdf *Exploiting Core Criticality for Enhanced GPU Performance*. (This paper discusses the issue within the context of single applications, sigh, but even large single applications can have multiple different kernels; and you can see that this only gets worse in the face of multiple different simultaneous processes. Even with their unsophisticated setup, taking criticality into account gets you 5% or so better throughput on average.)

Apple seems to have all the machinery necessary to handle this *outside* the GPU, with QoS tags attached to all NoC requests and honored at all routing points and by the memory controllers. What’s

not clear is the extent to which (if at all?) they monitor the kernels on each core for indicators of criticality, and then use the results to differentially tag outgoing GPU memory requests.

Volta is discussed here: https://old.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf
and here <https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf>

If you want extreme low level details of instruction latency and suchlike, you can read <https://arxiv.org/pdf/1804.06826.pdf%5B?url%5D>

nVidia Turing 2018

On the consumer side, the 2018 architecture is Turing. The name suggests something different from Volta but mainly, apart from the usual differentiators of high vs low end, the big new feature, justifying the name change, is ray tracing hardware, brand name RTX.

2018 is also the year the Microsoft DXR (Direct RayTracing) API is released and we start to hear about ray tracing everywhere.

ray tracing

Later we'll discuss the varying levels of ray tracing hardware, but what nVidia are willing to say is that they have hardware to support

- walking the BVH tree (Apple does this as of M1, M2, but using up the main shader hardware, not on separate hardware) and
- performing ray-triangle intersections (not yet the case for Apple, as of A16, though that's less important than the BVH support).
- also, of course, ray tracing denoising via AI via the tensor cores.

What nVidia don't say is that they don't (as of Turing) have hardware to handle recoalensing divergent rays. Divergent rays result in both divergent control flow (ie different lanes are running down successively different `if/then/else` paths) and divergent memory access patterns (ie the divergent rays all tend to read data from very different addresses, rather than all wanting to read parts of the same cache line).

Handling divergence is the harder part of the problem, and that has to wait till the 2023 architecture, Ada.

It's noteworthy that the ray tracing specific tasks of BVH traversal and triangle intersection happen on separate hardware, and so run in parallel with the GPU doing other work. Apple's solution, as of the A16, reuses at least parts of the generic GPU pipeline, limiting the amount of such parallel execution. With the A17 we get a scheme more like nVidia, with separate BVH hardware.

All in all this hardware runs raytracing about 10x faster than "equivalent" Pascal hardware, so if we take ~1.5x as a given for the Pascal→Turing transition, the Ray Tracing hardware is worth about a

factor of 6x. And that's about what we see (very roughly!) when we compare ray tracing performance on M1 to "kinda equivalent" Turing hardware. Apple claims the A17 is about 4x as fast as the A16 in "pure" ray tracing, which also tracks with these numbers, especially when you remember that the A16 had some specialist instructions that could perform parts of the BVH traversal faster than a naive FP32 shader, instructions that I don't believe were present on Pascal; so that the A16 presented a more performant baseline than did Pascal.

Left unclear is which, if any, in hardware Turing supports of

- building the BVH tree
- modifying the BVH tree to cope with changing geometry (apparently not)
- motion blur support for animation (as opposed to static images) (definitely not, we get this in Ampere)

As of 2018 and Turing the situation seems to be that ray tracing is fast enough to be used for image augmentation (eg more accurate shadows) but not for the creation of images/gaming from scratch with no traditional 3D pipeline involvement.

tensor core

A less expected change in Turing is that the Tensor cores are modified to also support INT8 and INT4 multiplication.

Perhaps surprisingly, Turing seems to get a full complement of Tensor cores, not a stripped down number. But only four, half the number, of load/store units in each SM quadrant, so half the local memory bandwidth of Volta. That sounds bad, but the caching system is improved enough that nVidia claim overall about 1.5x local memory bandwidth relative to Pascal.

A less obvious change is that FP16 (and vec2 FP16) operations are now routed to the tensor unit, not the FP32 unit. This opens up even more options for simultaneous execution of different operations, now allowing for FP32 and FP16 instructions to execute together.

In further good news, no more throttling of vec2 FP16 operations on consumer hardware!

We also see another, not insubstantial, boost in lossless memory (ie texture) compression – but still no utilization of lossy compression.

graphics

On the graphics side, nV talks about

- Mesh Shaders. The essential problem these solve is the construction and modification of geometry on the GPU.

There have been a variety of partial solutions to this over the years (first Vertex shaders, then Geometry shaders, then Tessellation) but these have all been limited in some way (or to put it differently, they have all tried to do most of the work on fixed function hardware). Mesh shaders seem like a final acceptance that the day of Geometry fixed hardware is over, and it's time to allow generic hardware into this,

earlier, stage of the pipeline.

I think this is why the Polymorph Engine, seen on earlier nVidia diagrams, is no longer present in Turing; that specialized hardware has been replaced by general purpose shaders.

Apple gets Mesh shaders as of Metal 3 and the A14/M1 generation of GPUs. It seems likely that Apple has likewise deprecated (or removed) its equivalent of the Polymorph Engine and dedicated Geometry hardware, but this remains unclear).

- Variable Rate Shading (essentially use lower quality anti-aliasing in regions where it won't be visible)

- DLSS, their advanced (and neural net driven) upscaling solution.

Recall that upscaling starts as purely spatial upscaling., based on filter theory with some attempt to detect and preserve sharp edges.

The next step, of limited success, and requiring some developer support so only really used in games, tries to also incorporate the previous frame into the upscaling.

DLSS claims to achieve better quality than this multi-frame combining, at improved performance (if you have a tensor core to do all the hard work!)

DLSS at this stage is still about more pixels ("4K quality while generating 1080p frames"). Thinking about it, you might also want temporal interpolation (ie "120 fps from 60 generated frames per sec") but that's not yet on the table. (nVidia research at this time is investigating synthetic slow motion, so clearly the construction of interpolated frames is on their mind; but it's not yet a product feature.)

- some technical modifications to Texturing.

Consider some triangle of a scene, with a computed texture drawn on top of it. Now consider the next frame. The triangle has probably shifted slightly, even if what we drawn on it (eg landscape, or fire, or moving water) is the same or similar.

There are two problems with our now re-shading this triangle.

- even if the computed texture is the same, we have to recalculate it because each sampling point within the triangle has moved;

- this movement of triangle relative to texture (except at unreasonable levels of anti-aliasing) will result in a shimmer of the texture. Turing allows for the shader, in cases like this, to draw the texture in a separate texture co-ordinate space. This allows the texture to be cached and reused (if it doesn't change frame to frame); and even if the texture does change, the calculation points for the texture computation remain the same, so the apparent aliasing and shimmer are substantially reduced.

Another way to think of this is that rather than computing the texture to fit the geometry of each frame, frame after frame, instead you compute a one-time "perfect" version of the texture, which is then reapplied, as appropriate, frame after frame.

The most interesting computational change to Turing is one that nVidia, for unclear reasons, has been very secretive about: the addition of *uniform registers* and a *uniform datapath*.

Uniform registers are scalar registers; think for example of a loop counter, which has no reason to be a SIMD/vector style variable.

This conceptually matches AMD's ISA which, although it has varied a lot, has generally provided scalar registers and a scalar data path separate from the main vector path.

The only place I have seen this discussed (and not much) is at a HotChips talk about Turing which includes the following diagram of a Turing quadrant:

We see that instructions can be issued to five end-points:

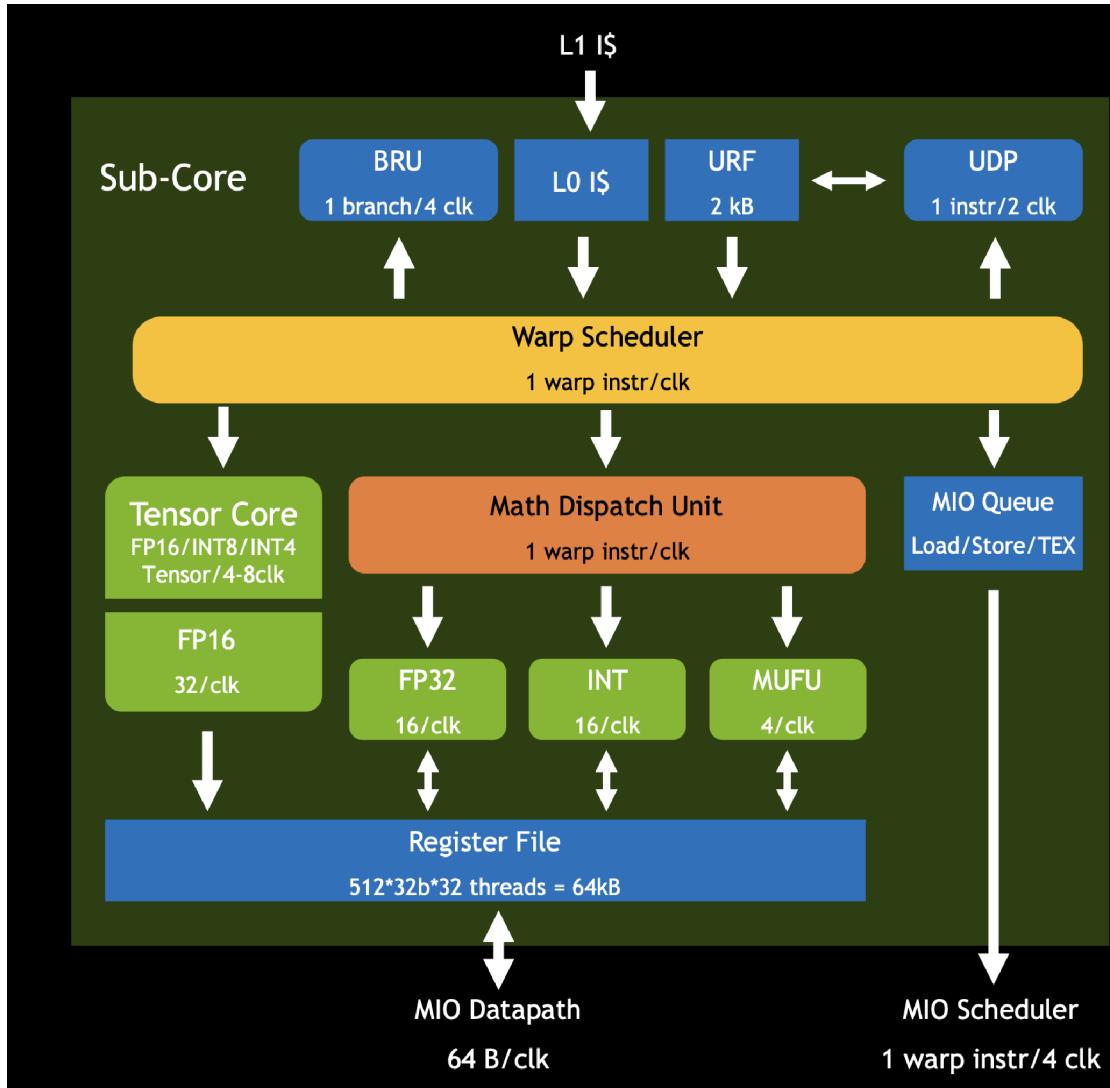
tensor/FP16,

"math"=FP32, INT, and special function,

Memory/IO (including texture),

branch handling, and

uniform execution (URF=Uniform Register File, UDP=Uniform Data Path)

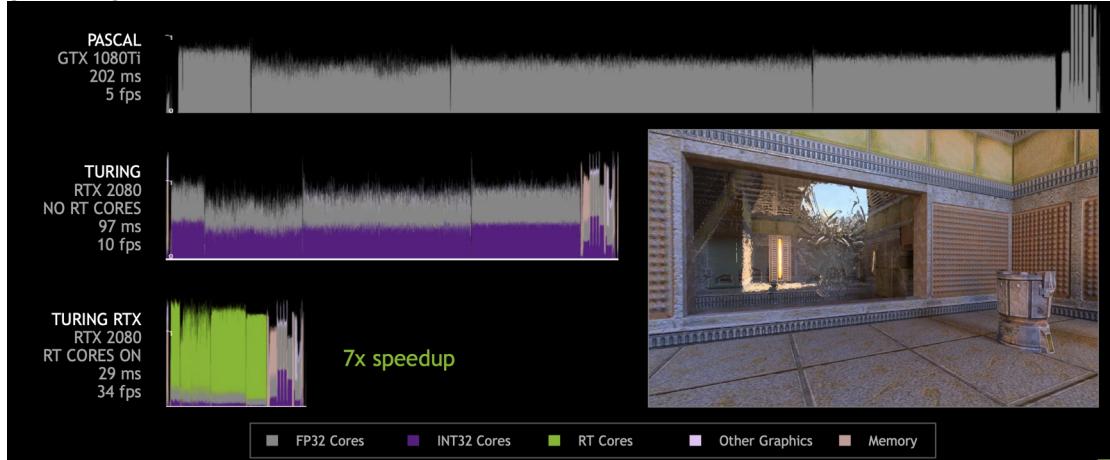


This builds on Volta's splitting off INT from FP32, so that FP32 units can work closer to 100% active, again allowing non-“primary FP” instructions to be shifted to a small unit performing alternate execution.

The claim is that these changes together (ie split FP32 and integer execution, and separate scalar/uniform registers and execution) boost the effective performance (“IPC”) of a core by more than 50%, and up to 2x.

Compare for example the graphs below:

Even apart from the boost of ray tracing, we can see that over the course of one frame, we manage to get very good simultaneous use of the INT32 and FP32 cores.



As a business point, ultimately NVidia produced a cut down Turing (the GeForce 16/GTX series) that removed the Ray Tracing functionality and replaced the Tensor cores with pure vec2 FP16 lanes. This doesn't have much relevance to our story except insofar as it shows that GPUs (rather more so than CPUs) can be redesigned in something of a cut-and-paste manner.

While Apple has not done anything like this, they *could* at some point do so, with a Pro GPU for the Pro/Max lines, a Consumer GPU for the A/M lines, and maybe an even more stripped down GPU for the watch?

(I keep raising this issues in different ways because it's clear that all the major GPU vendors have

found it easier to bifurcate their architectures than to maintain a single design optimized for all clients, from data warehouse AI training to STEM computation to 3D professionals to gamers; and it seems likely that Apple will face these same tensions.)

Turing is discussed here: https://old.hotchips.org/hc31/HC31_2.12_NVIDIA_final.pdf

and here: <https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf>

With a detailed dissection (mostly similar to the earlier Volta dissection) here: <https://arxiv.org/pdf/1903.07486.pdf>

nVidia Ampere 2020

In 2020 we get Ampere and, apparently, nVidia want to return to a single GPU design. But, honestly this toggling between separate brand names (Turing vs Volta) vs a common brand name (Ampere) is more marketing indecision than technical reality. We still get “data warehouse” chips with full Tensor core setup and no ray tracing, “consumer” chips with ray tracing and lower-end Tensor cores, and “low end” chips without even ray tracing.

If we judge by traditional GPU vector (FP32) performance, Ampere (at the highest end) is again about 40% “more” than Volta, and so a little over twice Pascal. But if we judge by matrix performance, then we’re at over twice Volta.

Not only is the performance of the Tensor cores substantially improved, they get even more formats, including BF16 and TF32 (a 19bit format that has wider dynamic range than FP16).

There are also various technical changes to the instruction set (to get more work done with one instruction) and to how the register file is accessed (so that accesses by the Tensor core take less time blocking accesses by the other shader cores).

The Tensor cores may seem uninteresting if you don’t care about AI training, but parts of nVidia are working hard to make them useful for other scientific work. Obviously you can use these for other scientific matrix multiplication, and to that end FP64 support has also been added, though only at about 2x the rate of doing it in the “shader cores”. But work is also beginning to investigate more unusual ways of using the Tensor cores.

For example suppose you try to run an iterative matrix solver by running the first few iterations at lower precision, then switching to higher precision. Let’s assume that you want FP64 accuracy. The summary is that you get some value, though not much by running early iterations in either FP16 or BF16, but you get a lot of value (average speedup about 2x) by running early iterations in TF32.

(Presumably you’d even more value running in FP32 if that were any option, eg if you were doing this using Apple AMX rather than an nVidia Tensor core. This sort of work is not just relevant to nVidia!)

There’s an interesting (and very Apple-like) optimization of barriers.

A standard barrier (think of a threadgroup barrier, for example) says two things:

- I have arrived at this point (meaning I have completed the work leading up to this point)

- I am waiting for everyone else to reach this point

There is no intrinsic need to tie these together. What you could do is something like

- call BarrierArrival: “I have arrived at this point”

- do whatever subsequent work the programmer can think of to fill in time (for example, preload data values you may need later)

- call BarrierWait when this subsequent work runs out

This makes barrier waits less expensive, to the extent that useful work can be filled in these intermediate time slots.

As an extension of the warp-wide Shuffles provided many iterations earlier, we now get single instruction warp-wide Reductions.

widespread memory/cache improvements

Memory bandwidth is doubled; as is NVLink bandwidth; L1D is 50% larger and the L2 cache is almost 7x larger.

The L2 cache is split into two physically separate partitions which are kept coherent. You can compare this to the coherence of two L2 cluster caches for the two P clusters on an M1 Pro.

This split obviously gives you an easier path to doubling (and more) the capacity and bandwidth of the GPU L2.

It also means that you want to add at least some smarts to your thread scheduler, to make optimal use of cache locality. And of course, the result is you have something that starts to look like an M1 or M2 Ultra GPU...

There's now something like Apple's DSIDs (data stream IDs) to control data residency in the L2 (ie you can effectively lock into the L2 data that you believe will be subject to substantial reuse in the future, but not the immediate future).

There is now generic lossless memory compression (not just texture compression), both in L2 (giving up to 2x the effective L2 size) and in DRAM. It looks like the DRAM compression uses a different (more effective, but presumably also higher latency) algorithm to get up to 4x compression.

Linking compression with the Tensor cores, sparsity support has been added, allowing for another performance factor of 2x for the Tensor cores, if appropriate. This is not as “obvious” as it looks! Everyone knows that neural network weights are sparse, the question is how can you, practically, make use of that? nVidia does it by imposing a “structured sparsity” in which 2 of each 4 weights are forced to be zero according to some layout pattern. This is obviously not general (in terms of what might be required, doesn't take advantage of >50% sparsity, etc) but is implementable, and general enough to be widely useful.

There's a GPU instruction to perform an asynchronous DMA-like copy into Scratchpad memory; a nice idea that avoids dirtying registers (that would otherwise be required for the copy) and dirtying the L1D, saves energy, and allows compute to continue while the DMA is happening. Think of an instruction something like `Load (n) BytesFromDeviceAddressR1ToStoreInSharedAddressR2 R1, R2` so that each lane can perform its own load/store (with no intermediate register) and the warp-wide set of loads and stores runs in the background while the shader cores do other things.

Something analogous to the familiar threadblock-wide barrier is called just before the point at which this (asynchronously copied) data needs to be accessed to block, if necessary, till the data is present.

scaling

On the “OS-like” side, *Multi-Instancing* is now supported as the next step in GPU virtualization. This is essentially the ability to split a single large Ampere into up to 7 “partitions” each with dedicated resources. So another way to look at it is as a (somewhat inflexible, but strongly isolated, and perhaps good for multi-client billing situations) way to enforce QoS. Compared to NVidia’s earlier attempts along these lines, we now also get hard partitioning of the L2 and memory bandwidth.

Within one of these hard partitions, earlier Volta-style sharing (MPS, aka Multi-Process Service, so much less rigorous QoS, shared L2, and shared memory bandwidth) can still happen, so you can imagine something like each department gets half an Ampere, then students can share kernels within their half.

Now suppose that you wish to execute a number of kernels with various dependencies amongst them. So kernel A feeds into kernel B, while kernel C is independent of A and B but D depends on both B and C. You can, in principle, express these dependencies by creating multiple separate queues, and submitting work appropriately to each queue, given that queues implicitly impose a sequential dependency, but separate queues can run independently. But this can be a hassle.

CUDA10 (2018) Graphs allow a nicer syntax for expressing this network of dependencies. But the Graphs syntax gives you more

- a single graph, provided as a one-time unit, allows for a “compilation” of the dependencies, to allow an optimal submission order of the tasks

- if (the usual case) the graph is executed repeatedly, just on different data each time, then the set of kernels can be stored on the GPU and resubmitted on the GPU without CPU involvement (as opposed to having to create and submit queues of tasks for every iteration)

- with Ampere the above resubmission has been made more efficient. NVidia doesn’t give details, but my guess is that the cores now provide feedback as to when kernels are close to completion so that the kernel dispatcher running on the GPU can start to prepare the next kernels to send out (eg prefetch instructions and perhaps some data).

Honestly, while Ampere seems to have generated very little excitement at the time, most of these ideas (apart from Multi-Instancing, which is of no interest to me!) impress me, and I hope Apple adopts their

equivalents soon.

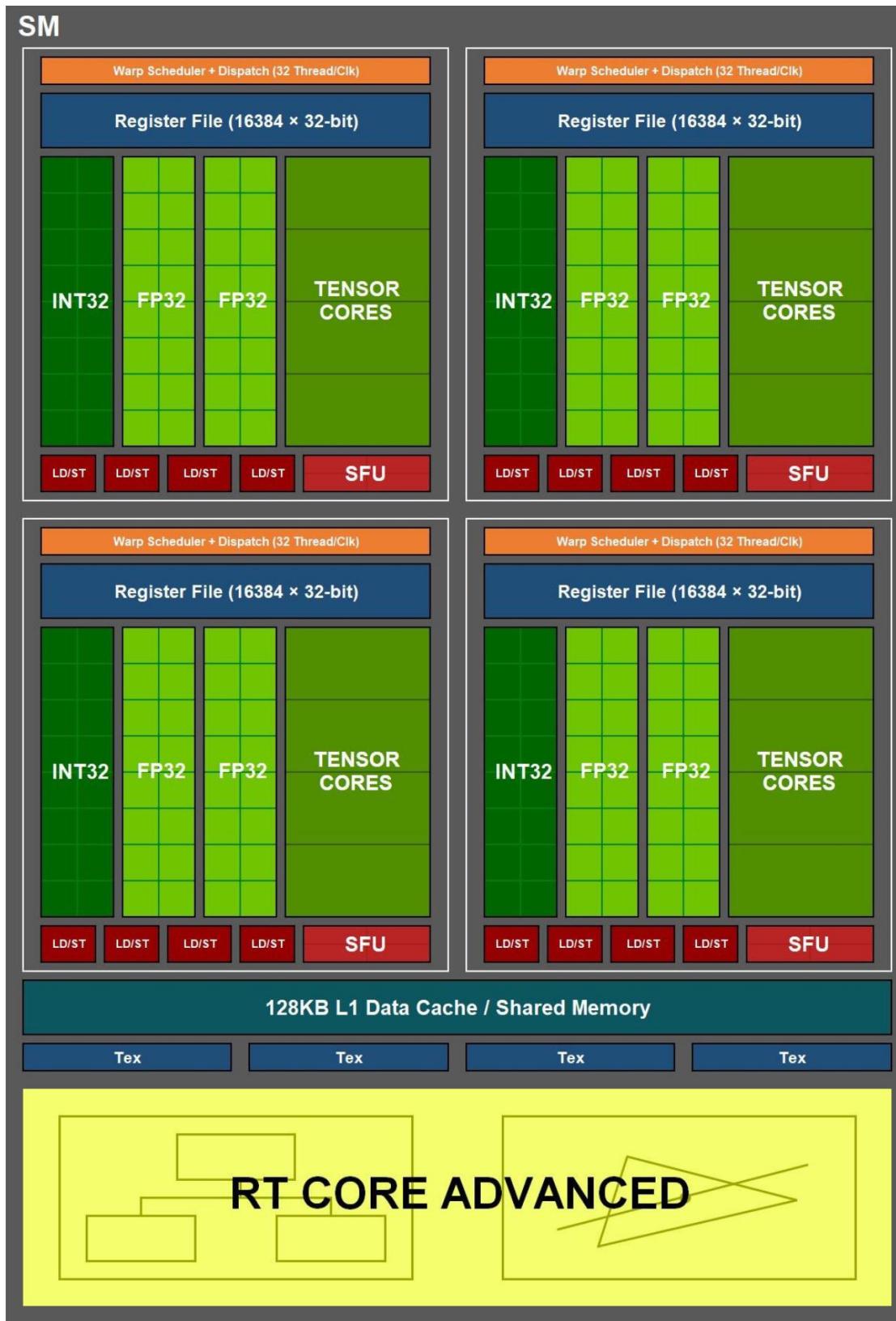
graphics

With Ampere we're back to a single name, but still a somewhat different consumer architecture. As expected we lose FP64 (and the obvious high-end features like Multi-Instance) and gain Ray Tracing.

More interesting is that the column of INT execution units becomes a column of INT/FP32 execution units. This might seem like a step backwards, but it makes sense insofar as the numbers showed we "on average" use INT about $\frac{1}{3}$ as often as we use FP32, and this allows us to somewhat rebalance usage. But still only one issue per cycle, though still with the double pumped cycling (units are 16 wide) so again "on average" we can issue two instructions over the time it takes to fully begin execution of a warp-wide instruction.

In other words one quadrant goes from

- one FP32/INT unit to
- separate FP32 and INT units to
- one FP32 unit and one FP32/INT unit (and a scalar unit).



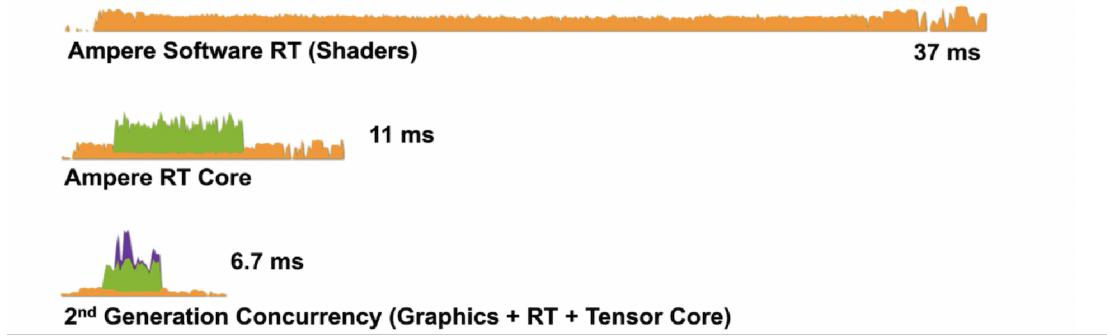
An obvious possibility at this point would be a shuffle network attached to register (or register cache)

read/write rather than as an execution unit. This could both handle

- the “how do I shuffle a 32-wide warp” problem, and
 - would allow you, if a divergent (ie branching) warp has 16 or fewer active lanes, to shuffle the source registers into 16 adjacent lanes and execute in one cycle rather than two.
- Does nVidia do this? Who knows, but if so they are keeping quiet about it.

Every year you think nVidia has become basically a pure compute engine, and every year you learn that in fact there was still some modality left in the design! With Ampere we learn that the Turing SM could handle a simultaneous graphics and compute (often ray-tracing) workload, but could not handle two compute workloads simultaneously! It’s unclear quite what “workload” means in this context (kernel?) but the example nV give is that on Turing ray-tracing could not execute simultaneously with tensor-core based denoising, and that is fixed (well, to the extent that these things are ever *really* fixed by nVidia!) on Ampere, by allowing concurrent ray tracing and compute. (So apparently still all manner of unclear limitation on exactly how many of what sort of kernel can run alongside other sorts of kernels...).

So we now get a diagram like this with three simultaneous “workloads” executing a single frame:



The other big change to Ampere Ray Tracing is support for Motion Blur. The way this is done (by everyone, not just nVidia) is as the leaf node of the BVH tree you no longer have just a triangle (against which you test intersection), you also have a linear expression saying how the triangle moves during the duration of the frame. Each ray is now emitted at a random time during the duration of the frame, and is tested against where the triangle would be at the appropriate time.

This all works very nicely, so well done nVidia; but it raises an interesting question: what’s the appropriate split between shader hardware and RT hardware?

By providing RT hardware, nVidia can run each ray tracing intersection test (which may be thousands of steps walking through the BVH tree) on separate hardware simultaneously with the shader core.

The alternative of running the BVH tests on the shader core means much of the time while ray tracing, your expensive (lots of FP32 hardware!) shader is mostly sitting idle or doing simple memory lookups. But RT hardware sits idle if you don't care about RT tasks!

Can we reconcile these two?

One possibility might be to make RT hardware become a generalized engine for walking data structures. Lots of memory hardware, some simple integer hardware to calculate addresses, perhaps some limited FP comparison hardware. The engine would be programmable with a "Walk" shader that describes how to walk a data structure, and a "Node" shader that describes what to do when you get to a leaf node.

No-one seems to have done anything like that yet, but it seems an obvious next step. After all, where throughput computing is headed is a combination of compute on final nodes, along with traversal of complex data structures to reach those nodes. Think of adaptive grids for CFD, or sparse matrices, or fast multipole methods, or heck even the FFT.

A perfect throughput engine would exploit the technology of GPUs for collecting the data from memory (many threads, ability to sleep a thread waiting on memory) while **not** forcing expensive FP resources to sit idle on the shader cores that are performing this memory traversal... Imagine replacing that RT Core Advanced block at the bottom of the Ampere diagram above with something like two stripped down quadrants that have partial (not full) register files, INT and load/store but minimal FP cores, and which maybe even executed a simplified instruction set. Designated clauses in the instruction stream would run (optionally asynchronously) on these data engines, which could perform the equivalents of search, copy, and prefetch, across complex data structures, transferring the results via L1 or Scratchpad. This gives you all the value of the RT Core, but in a form that's useful to all developers.

Another change that's not exactly hardware, but an interesting direction, is that nVidia has implemented some high throughput lossless decompression algorithms on the GPU. This is in addition to the lossless memory compression stuff, or the earlier texture compression stuff. This is called *RTX IO*.

The immediate use case is that compressed game assets of a variety of types which previously would be decompressed on the CPU (a process which slowed game launch) are now decompressed on the GPU; but one can imagine this also being of use wherever GPUs are required to scan through larger-than-memory datasets. A rough version of the same idea is that nVidia includes a small JPEG block on at least some of their high end chips to speed up training against vision data sets.

You can read about Ampere at https://hc32.hotchips.org/assets/program/conference/day1/HotChips2020_GPU_NVIDIA.Choquette_v01.pdf

and

https://www.megware.com/fileadmin/user_upload/LandingPage%20NVIDIA/nvidia-ampere-architecture-whitepaper.pdf

nVidia Hopper 2022

nVidia is now totally invested in AI, so it should come as no surprise that the headline change is an update to the Tensor Core, to improve the performance of Transformers. Technically what this appears

to boil down to is that

- statistics are captured during the processing of each layer of a neural net
- the compiler, knowing something about the structure of transformers, includes calls to capture these statistics for each layer of the network, and emits code variants that, based on these statistics, computes each subsequent layer as either FP8 or FP16.

Of course this implies that the Tensor engines now support FP8, and that's correct; in fact they support two slightly different versions of FP8.

We get the usual increase in memory bandwidth, cache sizes, etc; nothing new there. The one twist in this area is nVidia, learning from Apple now providing a pairing of the Hopper GPU with their Grace CPU to provide an optimized CPU↔GPU linkage. You can see this as nVidia losing patience at being slowed down by the incompetence of their CPU partners (ie Intel and IBM) and deciding they can do a better job internally by controlling the entire package from CPU and CPU RAM to GPU to joint circuit board.

One wonders if, over the next few years, first the “Pro Visualization” cards, and then even the consumer cards, will get a dedicated CPU on them? It seems to me this is not a completely crazy idea, even if this is mainly used to split the driver into a small host section that mainly interacts with the primary OS and app, and a much larger section that runs on the nVidia CPU and does the heavy duty driver computations.

A very general extension to Ampere’s `LoadFromDeviceMemoryStoreToSharedMemory_Async` instruction is provided, now with the ability to asynchronously copy a sub-tensor embedded within an up to 5D tensor, specifying the dimensions of the source and embedded tensors, and having the async hardware perform all the relevant address calculations as well as the data movement.

There are some new instructions provided for accelerating Dynamic Programming. These are essentially fusions of three simple operations, things like `max[min(a+b, c), 0]`. They look like (and are!) pretty specialized, but very helpful for comparing sequences (DNA, or proteins, or, probably also text), or for certain path planning tasks.

The next step in nVidia’s compute vs graphics bifurcation is that almost all graphics support is now stripped from Hopper, with what remains present probably only for backward compatibility and likely to be removed in the next generation.

threadblocks

Less glamorous, but probably most important long term, is a complete revision of the baseline idea of a threadblock, matching the earlier rethink of the idea of a warp.

For decades a kernel has consisted of threadblocks consisting of warps consisting of lanes; and the threadblock is the basic unit of co-operation.

But threadblocks are small (~32 warps); so it would be nice if code could co-operate (share data, synchronize rapidly) at a larger granularity, to match now much larger GPUs.

We get this with *Threadblock Clusters*. A cluster is a set of threadblocks that execute together, utilize shared Scratchpad, and can rapidly synchronize.

When you first hear about this, this might seem to dramatically limit the flexibility of GPU scheduling; but if you think about it a little longer, it's not so bad.

If you get confused, think about what I describe below in terms of the easiest case, a cluster consisting of a pair of threadblocks.

The life of a threadblock consists of

- coarse-scheduling, which places the threadblock on a GPU core (ie an nVidia SM) and allocates the desired Scratchpad storage for the threadblock
 - intermediate-scheduling, which activates warps from the threadblock, issues instructions from them, and puts the warp back to sleep when it waits on memory
 - coarse-scheduling, which notifies the outermost scheduler that this threadblock has completed its task

Only the coarse-scheduling has to be modified, to ensure that not just a threadblock but an entire cluster are activated together and acquire the full complement of Scratchpad storage (spread over multiple cores) that they desire. After that everything schedules as before.

For this to work we do also need

- rapid communication paths between the L1/Scratchpad storage on separate cores. Of course this is not the same thing as (and is quite a bit cheaper than) making the L1's coherent.
- a way to enforce rapid synchronization across cores (including executing atomics from one core in the Scratchpad of another core)

But the end result is that we can now work, within highest speed storage, on much larger units of work than before!

My guess is that the largest cluster supported is half a Hopper (matching the way Ampere was split in two, with each half having a separate L2, the two L2's being coherent); but nVidia is not sharing such details.

NVidia call this Distributed Shared Memory [Shared Memory is nVidia's term for Scratchpad], and I'm guessing what it boils down to for developers is something like (if you opt into clusters) an expanded address range for Scratchpad that scales linearly with the size of the cluster.

Documentation includes

<https://www.servethehome.com/nvidia-h100-hopper-details-at-hc34-as-it-waits-for-next-gen-cpus/>

and

<https://resources.nvidia.com/en-us-tensor-core/gtc22-whitepaper-hopper>

With 2023 we get Ada Lovelace, the updated consumer GPU. Significant new graphics features include

- The ray tracing engine includes optimized alpha support, ie support for only partially transparent triangles. This is not exactly what you might think and is in fact cleverer than you might think.

Essentially it's a way of augmenting geometry which includes fully opaque, fully transparent, and partially transparent portions. The geometry is augmented with a fine map (understandable by the ray tracing hardware) which describes the opacity of each piece of the geometry. The net result is that, where previously any ray that hit the geometry would have to call a shader to decide how to handle it, now the two simple cases (fully transparent and fully opaque) can be handled rapidly and only a genuine partial opacity hit requires a call to a shader.

- Complex geometry results in a complex BVH tree, very expensive to create and somewhat expensive to traverse.

Reuse of the same sort of hardware as the Opacity Micromap engine described above allows for on-the-fly construction of something like a bump map within the ray tracing hardware. This allows for a much simpler, rapidly generated BVH tree, where after the final node is reached the fine-grained geometry is constructed on-demand to test triangle intersections.

- Shader Execution Reordering, which is reordering of divergent rays (within a threadgroup? a thread-group cluster?) so as to reimpose convergence.

On the one hand, good for nVidia for implementing an idea that's been much discussed (but only discussed) in the academic literature. On the other hand, like many (not all, but many) of the most recent nVidia advances, the implementation looks narrower than one would like, a quick solution to one particular problem, rather than generic machinery that will reconverge divergent execution paths for generic GPGPU execution :-(

- Use of a hardware Optical Flow Analyzer to detect the motion between frames and use that to inform DLSS upscaling (now including, at least that's the way I understand it, temporal interpolation), so that DLSS3 can generate say a 60fps stream from a 30fps source.

You can find some details here <https://images.nvidia.com/aem-dam/Solutions/Data-Center/l4/nvidia-ada-gpu-architecture-whitepaper-v2.1.pdf>

Conclusions from this history

conceptual conclusions

What can we conclude from this historical overview?

Essentially by the mid-80s everyone agreed what a CPU should do. It should be based on bytes, should support a 32-bit address space (with 64-bits as the obvious next step), should support paged memory, interrupts and pre-emption, IEEE floating point etc.

By the late 90s it was even pretty much agreed the best ways to build a CPU (OoO, speculation, caches, etc), and since then everything has been refinement and optimization to the details of new processes.

GPUs lagged this quite a bit. There seems to be agreement over the past few years that a GPU should provide an MMU (and all that implies in terms of security, allowing for a common address space with the CPU, etc); that pre-emption should be available; that a Scratchpad should be provided along with transparent (but not especially coherent) caching.

But much remains not agreed upon! Should a GPU just be expected to support FP64 well? Or even at all? How about FP8? Should the FP support be full IEEE? Should the lanes be independent enough to support per-lane synchronization/locking primitives?

And you could imagine additional functionality that no-one provides – but perhaps they should?

Perhaps it makes sense to allow a developer to supply their own scheduler that decides how to split a grid into threadblocks, how to distribute threadblocks across cores, and when to swap between threadblocks on a core?

Likewise in terms of how to implement GPU functionality there's some agreement, but also a lot of divergence. It seems like everyone has converged on a unit (call it a “core” consisting of four SIMDs [a set of 32 execution lanes]) so that these four quadrants share L1I, L1D, Scratchpad, and some other caches).

But what to put in a quadrant? nVidia put Tensor cores, Apple instead does matrix multiplication using the pre-existing lane ALUs and some data routing. Should we have a scalar lane as well as the 32 SIMD lanes, to handle scalar code like loop counting? Should we go for maximal simplicity (in-order one instruction per cycle dispatch) or some degree of sophistication (try for eg two instructions per cycle dispatch, maybe even some limited degree of OoO?) Should we put almost all the scheduling work in the compiler/driver (as nVidia does), or use basic score-boarding to inform the dispatcher when an instruction is safe to schedule?

Even the caching schemes are unsettled. Apple and nVidia both appear to use a two level scheme, where each core has a non-coherent L1D (which is, at certain kernel boundaries flushed into a global L2, so that from kernel to kernel we see coherence, but not within a kernel); this is extended upward in the largest designs by splitting the GPU in two, and requiring the two L2s to maintain traditional coherence.

But AMD uses a three level scheme. Each core has an L0D, then a group of cores (currently about five cores) share an L1D, which is flushed out to a global L2. AMD says this scheme allows for better bandwidth – the L1D's handle enough traffic locally that their L2 doesn't have to be as high bandwidth (eg have as many wires converging into one place...) as the Apple/nV L2. Is this overall a good idea?

Which means it's basically hopeless to assume any sort of convergence at even lower-level implementation details!

One reason I mention these issues is that once we start to examine the Apple design in detail, we will encounter various details in the patents that don't quite make sense (eg the so-called *masters*, which I think refer essentially to fixed function [ie non-programmable] hardware; or the so-called *clauses* (a way to organize the GPU instructions). We'll try to understand these, but it's quite possible that they're

simply no longer important; relics of an earlier design space where they made sense, but abandoned as the GPU evolved to become ever more general purpose, and with ever more transistors available to it.

When it comes to optimizing your GPU, which will inform much of what we read about the Apple design, the thing you have to remember is that ultimately you are gated by energy/power. You may decide your power budget is 8W, you may decide it is 400W; either way it's easy to exceed the power budget, meaning that increasing performance is *primarily* about reducing energy – which frees you to do more of what you want. Area is a second order limitation, but behind energy.

Next point is that (as always) moving data is most of what costs energy, not computations. Which means that your highest priority is always reducing data motion as much as possible. Just like the CPU, you're obsessed with caches and avoiding trips to RAM; but the details are very different. The CPU is all about reduced latency and a general expectation that once you stop using data it's unknowable when you might reuse it; whereas the GPU cares more about bandwidth and throughput, and you can substantially predict data reuse – but that reuse will be a 30th or 60th of a second later, which a lot of data streaming through the GPU in the meantime.

This means that

- large near caches (L0, L1, even L2) probably don't provide much value. They can't hold the large data long enough for reuse, and the data that is constantly reused within a single frame is small. Make them large enough to hold that small amount of constantly reused data, and "overlapped" data (ie we load in a cache line whose data is then loaded by multiple lanes, possibly even different warps within the threadblock; likewise to aggregate writes, ideally to cover a full cache line before that line is written out), but no larger. L2 also acts as a communication hub between cores so optimize for that as much as for capacity.
- SLC probably *can* store much of the data that is reused from frame to frame, so provide technology that allows that to happen (eg ways to tag lines so that they are or are not preferentially retained in SLC, and ways to flush lines corresponding to resources no longer in use).
- GPUs can calculate a lot of intermediate data that is used later in the frame (eg temporary buffers used to approximate shadows or global lighting or whatever). Provide mechanisms to ensure, as much as possible, that that data stays local; and that, after we're done, it can be forgotten locally, without ever even having to be written out to a higher level of cache or DRAM.

This same thinking extends even down to the register file. GPUs have large register files (to keep as much computation local as possible) but this means reading from that register file is expensive and slow. So provide some sort of caching mechanism to try to hold active registers even closer to the execution units, along with machinery for the compiler to make maximal use of this. (Indicate that some registers being used by this instruction should be retained for reuse by the next instruction; indicate that these other registers are now dead and can be removed from the cache; etc) You can also try to get the compiler to help with this; eg if a computation constructs temporary values, try to write those temporaries to otherwise dead registers that are already present in the cache [try to maximize local register *name* reuse, not just value reuse].

Can we do a better job of using L1 caches? The current state of the art is essentially to allow L1 cache use while a kernel executes (with no promise, in the absence of fancy fences) that any threadblock and hence data in one particular L1 will see changes made by another threadblock in a possibly different L1; then flush and invalidate the L1 caches out to L2 at kernel boundaries. One can work on making this more efficient (eg limiting what has to be flushed out as opposed to what can simply be invalidated). Even better would be if, somehow, we could tag data in the L1 cache by the respective “client”, so that we only had to perform this for the data owned by a particular client, not everything in the L1. Best of all would be if we could somehow (in a lightweight enough fashion) continue to reuse data in a particular L1 if subsequent kernels corresponded to the same “client” and could make use of the data already in the L1.

The academic state of the art in this appears to be the De Novo protocol, (2015) <http://rsim.cs.illinois.edu/Pubs/15-MICRO-scopes.pdf> *Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models*, but it seems like no-one commercially is yet using anything like this, though Apple appears to be using ideas in this sort of vein. xxx check

There is a slogan in GPU computing: “Performance=Parallelism; Efficiency=Locality”. Parallelism is obvious in the sense of multiple chips (eg Ultra) of multiple core of multiple lanes. We’ve seen ways nVidia has taken this further, to split a core into multiple quadrants each executing separate streams of instructions, and then to trying when possible to have multiple execution units within a core executing simultaneously (something like superscalar execution), along with the RT core usually set up, then executing for a few thousand cycles independently of the rest of graphics.

So something to take note of in the evolution of the Apple design is the extent to which they try for the same goal (every piece of hardware in the GPU always active, not just the more obvious goal of “able to execute simultaneously multiple lanes of multiple warps”).

Likewise efficiency (ie reduced energy usage) is all about locality and we’ve seen something of how nVidia handles this with various caching levels and (somewhat limited) schemes to tag data flows, trying to control their cache persistence.

Take note of how Apple is tackling the same concerns in ways that are similar to nVidia but often just a few steps more complex/sophisticated.

If you want to think how both Apple and nV will evolve their hardware, this slogan is a good starting point...

future product conclusions

One thing that should be clear from this review of nVidia’s recent history is that nVidia is not ARM, let alone Intel. They have not sat on their laurels in the slightest!

Every design iteration includes not just nice tweaks to make existing hardware more efficient, but substantial rethinking of what a GPU can and should be. Most recently we’ve seen a substantial rethinking of what warp execution should mean (allowing for dramatically new types of algorithms), and a rethinking of how better to exploit “locality” given that GPUs now include so many cores, each with their own Scratchpad storage.

We'll see that while Apple have done pretty well in maintaining pace with nVidia (always with their own specific concerns and optimization tasks, just as nVidia has their specific concerns) they have also fairly robustly stayed four to five years behind nVidia in terms of the functionality that's most important for designing new ways to use the GPU. While a basic GPU developer has no reason to complain about what Apple provides, leading edge STEM GPU researchers basically have no competition for nVidia; it's only nVidia that's shipping today (as opposed to in five years) the best hardware for designing very new ways of using throughput computing.

If there's one thing that might give Apple comfort (and it's a weak reed indeed), it's that nVidia's Hopper and Lovelace documentation are even more marketing-speak than their earlier Ampere communications, and that both the Ampere Hot Chips talk and white paper were clearly written by the marketing department, unlike the earlier engineering-written documents.

Does this mean the marketing parasites are starting to worm their way into nVidia's brain, destroying the competence of the past 30 years? Who knows? People like Bill Dally or Jensen can't patrol for competence everywhere, and once the parasites take over an organization, they're near impossible to dislodge – look at Intel years after Pat insisted he was going to change the culture. So, we shall see. Blackwell will be out next year; let's see what that's like, both as technology and how that technology is presented to us.

Let's consider some other general points, a few of which are reiterating what I've already said:

- Bill Dally (chief engineer at nVidia) claims that the “full” cost of performing a traditional FMA on a GPU is about 20x the cost of the actual floating point work. The overhead is (small amount) reading, decoding, and moving around the instruction and (much more) reading register values from SRAM, moving them around, and writing them back. The consequence is that the richer you can make an instruction, eg - a matrix multiply instruction, or
 - a fused instruction that does multiple operations as one instruction, or
 - a single instruction that moves around a lot of data without having to copy it 32bits at a time through the register file, or even
 - (to some extent) a SIMD2 or SIMD4 instruction
- the more energy you can save.

You can see nVidia's specialized Dynamic Programming instructions added to Hopper in this light.

- It's natural to think in terms of the efficiency of use of hardware area (and this is certainly my default mode of thinking!) But power density may be your most important constraint, meaning that you have to be very careful that even good ideas for more aggressive execution (eg superscalarity, or OoO) are extremely low energy overhead.

- Multiplications tend to be quadratic (both area and energy) in the length of the operand. Hence the constantly shrinking operand lengths available, especially for the tensor units (FP16, TF32 [which is really just a slight tweak on FP16 for multiplication purposes], even FP8). Apple have gone a slightly different direction with this, concentrating on using short indices for weights (so that rather

than having a weight be, say, FP8, the weight is a 6- or 4-bit index into a table of FP16 values. This table is an optimized quantization of the raw weight values of the model. Apple's solution probably(?) saves more energy, in that less data has to be moved around, but doesn't save execution unit area, or energy for the actual multiplication. Win some, lose some!

It's always easier to see what a company does than what it doesn't do... What are prominent concepts that seem to be missing from nVidia's resume?

One item I notice is the handling of point clouds, something important for AR, and something where Apple has a few patents (for example ways to compress point clouds). Another is handling of voxels, where nVidia doesn't seem to have ever provided any specialist HW support.

Research points

Given all the issues we've seen, let's now think about various decision points in how we might design a GPU, after which we can try to see what Apple does.

Our starting point is that the GPU (or at least the GPU driver) is aware of a pool of kernels to be executed. Some of these kernels must follow others, some are independent of others. Even at this point we have options

- should we, as much as feasible, try to finish one kernel before beginning the next or, alternatively
- should we try to interleave some large number of kernels to be running simultaneously on different cores, and even on the same core.

This boils down to a question we will see repeatedly:

- what are "resources" required and desired for executing a kernel, threadblock, and warp? The required resources is easily answered: to begin a threadblock requires we have enough Scratchpad available on the target core; to begin a warp requires we have enough registers available in the target "entity" (perhaps a core, perhaps a core quadrant).
- but different kernels may also make heavy (or light, or no) use of compute (FP64 or FP32 or FP16) vs eg specialized graphics hardware (texture hardware, ray tracing hardware), or L1 capacity/bandwidth, or L2 (or SLC, or DRAM) capacity/bandwidth
- kernels might also have associated QoS

In a perfect world, the kernel-level scheduler would know all of these features and would schedule kernels based on them. In such a world it would make sense to try to finish (with no sharing) a kernel that makes heavy use of memory resources, but it would make sense to share with a different kernel (if available) that mainly makes use of compute resources.

How close, then, and how, can we approach the ideal?

Starting from the other end, what's the simplest we can get away with?

The simplest is to execute whole kernels, in order, no sharing, perhaps via first come first serve.

As soon as we go beyond that we have the following situation:

- suppose kernel B depends on kernel A, while kernel X is independent.
- then it's suboptimal to schedule kernel B after kernel A (even if our target is more or less one kernel at a time) because this means that we have to wait until every aspect of the work associated with A is cleared out of the machine, every last straggling warp, every last memory flush, before we can start B. It makes more sense to get X ready to run as soon as A no longer uses up the entire GPU, and then schedule B as soon as X is in that same position of no longer dominating the entire GPU.

This is essentially the nVidia policy as of 2020 (Turing) and perhaps still, along with an unsophisticated handling of priority in that higher priority kernels (if present) will be chosen over lower priority. This policy seems to be called the *leftover* scheduling policy, which I guess refers to the idea that once we start a kernel, we keep handing blocks as long as any are leftover before we start the next kernel. (Along with the addendum that we make the obvious decision to then fire up an independent kernel rather than forcing the delay of sequential dependence.)

Of interest is that it makes no attempt to interleave kernels that might, possibly, run well together each using different resources.

Next up, a kernel is built of threadblocks. But what determines the shape of those threadblocks? For example if my grid is a square of 1024×1024 points, I could define a threadblock as 32 warps in the x -direction and 1-high in the y -direction, so that it is long and skinny. Or I could define a threadblock as 32 warps in the y -direction so that it is close to (in this case perfectly) square.

Which should I choose?

As far as I can tell, nVidia requires that you tell CUDA the (x, y, z) dimensions of your threadblock (subject to the total size, ie $x \times y \times z \leq 1024$). Apple allows you to specify a maximum size, but not the (x, y, z) dimensions. So which is better, long and skinny or as close to square as possible? In principle it depends on how much data we expect to share at the edges between threadblocks, and the extent to which adjacent threadblocks route to the same L1 and L2.

Once you decide on the size of your threadblocks, how do we peel them off from the grid? Even with a 1D grid, we could start at the high end and go down, or at the low end and go up. Or we could divide the entire grid over the number of cores, and then feed each core a contiguous stream of threadblocks. The options only increase in two or three dimensions.

Once the higher level scheduler has a pool of threadblocks known to it (from one or more kernels) how does it decide to send those out for execution?

The easiest option is static allocation, so we divide up the grid (in some way) relative to all the cores, allocate each grid fraction to each core, and go from there. Obviously this behaves sub-optimally if some cores finish a lot sooner than others.

Alternatively we could perform “push” dynamic allocation, where the scheduler maintains a queue of pending threadblocks, is sent a one bit interrupt by a core asking for more work, and gets sent the next threadblock in the queue. But even this is far from optimal because neither side is making an informed decision as to the next threadblock sent out. In the worst case, this would result in, for example, a threadblock being sent out that cannot yet begin to execute because it needs to allocate more Scratchpad on the destination core, and that can't happen until another one (or more) executing

threadblocks on that core complete and free their Scratchpad.

nVidia (again as of Turing) seems to use a very slightly tweaked version of this. Something like: when a core sends a request to the Thread Block Scheduler, it sends not just a one-bit request but how much Scratchpad free space it has available. The Thread Block Scheduler then (apparently at some fairly coarse granularity, not immediately) schedules threadblocks to cores that will have enough space to execute them, possibly sending two (or more) threadblocks if they will fit, a so-called *most room* policy. The paper (2020) <https://www.samogden.net/assets/pdfs/Gilman2020a.pdf> *Demystifying the Placement Policies of the NVIDIA GPU Thread Block Scheduler for Concurrent Kernels* talks about this, though I find it unsatisfactory because it frames the questions, and thinks about its results, very differently from how I see the problem.

The important point I think, regardless of timing details, is that the Thread Block Scheduler is aware of the amount of free Scratchpad on cores and uses that in deciding where to route threadblocks, and how many; but doesn't seem to take anything else into account.

Once a threadblock arrives at a core, then what? An obvious optimization would be that a core can buffer a threadblock, so that we can queue up at least one excess threadblock on every core, and begin executing it as soon as the previous threadblock ends, rather than waiting until we get sent a new threadblock or few from the Thread Block Scheduler. But I don't know if anyone does that.

Next we have to allocate resources (ie the required Scratchpad storage) to the threadblock. At that point on this core we can then start to hand out warps. Once again, we can only begin a warp if there is enough register storage available; if not we will have to wait until at least one executing warp completes. Again there is scope for some degree of optimization here, perhaps, in terms of things like not bothering with the threadblock allocation until we can also simultaneously begin at least one warp? There are also decisions that could perhaps be made as to which quadrant to route a warp. We have seen that nVidia is inflexible on this, but nVidia's philosophy at every stage seems to be to do the simplest thing you can get away with, and make up for the inefficiency and delays by just having more hardware. If you go to the other extreme of working hard to choose optimal collections of different threadblocks from different kernels that work well together, you probably want to retain that flexibility all the way down to allowing any warp to execute one whatever quadrant of a core happens next to become available.

The literature in this space is somewhat unsatisfactory. Detailed GPU models are rapidly obsolete, the main thing IMHO one can learn is whether a given intuition is or is not valuable.

(2009) <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=d637974e983989271e3550d41c8fde582ab9da15> *Enabling Task Parallelism in the CUDA Scheduler* shows that substantial throughput improvements are available by running two kernels simultaneously, especially if they use different resources (memory vs compute bound).

(2014) <https://arxiv.org/pdf/1406.6037.pdf> *Preemptive Thread Block Scheduling with Online Structural Runtime Prediction for Concurrent GPGPU Kernels* imagines an extremely simple modification to the basic first come first serve nVidia scheduler, and suggests that even this provides ~20% throughput

improvement.

(2019) <https://dl.acm.org/doi/pdf/10.1145/3326124> *Coordinated CTA Combination and Bandwidth Partitioning for GPU Concurrent Kernel Execution* is now sophisticated enough to understand that we need to be careful about the types of threadblocks we co-execute, so that for example bandwidth dominated threadblocks don't hurt latency sensitive threadblocks. These papers start to suggest that we

- monitor resources so that the schedulers have at least some clue as to what's going on
- together with various combinations (depending on the exact paper) of scheduling at the warp, thread-block, and/or kernel levels to ensure better-than-random resource combinations running on the GPU as a whole, and within each core.
- they make the even stronger statement that knowledge, and optimal combining, is still not enough; in a perfect world we'd also utilize this information to attach QoS info to the transactions of each warp so that the high-bandwidth kernel and low-latency kernel each get handled optimally by the cache and memory systems. The claim is that using QoS correctly in this way has non-trivial consequences, boosting throughput by ~10% on average, and up to ~50% on pathological kernel co-executions over even the best possible threadblock scheduling (which then fights it out over the caches and NoC without QoS tags).

(2016) <https://pdfs.semanticscholar.org/387d/5b24317395ae7a86c8ecc9403ac62ed6febe.pdf> *LaPerm: Locality Aware Scheduler for Dynamic Parallelism on GPUs* looks at scheduling from a different angle, concerned with data reuse between threadblocks and across kernels (especially kernels that launch sub-kernels). They see substantial amounts of such data reuse, meaning that it's worth optimizing for it in every possible way, via related threadblock co-execution in time and on the same core, and in the ordering of kernels, so that child kernels run concurrent with or right after their parents.

This is updated and “rationalized” in 2020 <https://dl.acm.org/doi/pdf/10.1145/3406538> *Inter-kernel Reuse-aware Thread Block Scheduling* which takes the above lessons to heart, but also considers how to balance the situation in irregular algorithms where some threadblocks, sub-kernels, and kernels may finish a lot sooner than others; one has to trade off executing on the same core (or even same sub-GPU with common L2) vs taking advantage of a newly available core or sub-GPU without the same memory locality.

The earlier papers can be skimmed; this one is worth reading more carefully being both more modern, and at least mentioning various side issues along with the primary purpose of the paper.

If you want to get very technical, then the sophisticated buzz word regarding this scheduling (at every level, from kernels to threadblocks to warps to even individual lanes) is *progress* (and specific guarantees that are or are not made regarding the scheduler). As we've already mentioned, the state of the art for nVidia is enough flexibility in each lane that essentially any “reasonable” algorithm should make progress, Or should it? As of Volta, lanes within the same warp and threadblock can move relative to each other, but kernels are very large. Suppose progress in one threadblock requires progress in a

different threadblock far away? Are the higher-level schedulers flexible enough to pause a threadblock that is waiting, move it **off** a core, start the far away dependent threadblock and run it for some cycles, then revert to the paused and waiting threadblock?

That seems a heck of an ask... We expect that on a large machine (perhaps a few thousand threads) the OS scheduler will eventually run every thread, so that deadlock-free code will make progress. But is it reasonable to expect every thread(block) to eventually be run on a GPU with hardware scheduling and juggling perhaps hundreds of thousands of threadblocks?

The difference is that we expect a CPU to “loop” its scheduling so that any thread that blocks eventually gets swapped out, moved to the back of the queue, then swapped in again. We are not so quick in expecting a GPU to “loop” its scheduling so that a threadblock that blocks is eventually moved off core, placed at teh back of a queue, then later swapped in again – that doesn’t fit with anything we’ve been discussing about how threadblocks are scheduled, peeled in a geometric order off a grid, submitted to a core, and executed to completion...

There is some discussion of these progress issues and what is (slowly) being formalized here: (2023) <https://www.youtube.com/watch?v=g9Rgu6YEuqY> Forward Progress Guarantees in C++ - Olivier Giroux - CppNow 2023.

A similar research issue is how to handle “coherency”. As you probably know, this can be discussed in a variety of ways, and in recent years (last 20 years or so) rather different ways of talking about this have come into fashion. I’ve discussed the issue in the CPU section but let’s do a quick recap. The formal way of thinking about this (which is what we see in C++ language), but which I will still describe informally, is

- we split all memory accesses into *data* vs *synchronization* accesses
- within a singe thread accesses are ordered in the obvious program order way
- across threads all *synchronization* accesses need to be specially indicated; they then impose ordering on the prior accesses within the same thread.

Various operations then act as “publish” operations (act as an ordering barrier so that everything before this operation must be made visible to all threads before this last [write-like] operation) or as “subscribe” operations (this [read-like] operation must complete before all subsequent operations). We earlier described how and why this works in terms of storing various (eventually to be shared) data, then as the last stage writing a “valid” flag indicating the stored data has been finalized; and likewise on the read side.

The way this plays out in practice has multiple pieces because of history and compatibility, but the basic idea is that we call the write or publish style operations *release* operations, and the read or subscribe style operations *acquire* operations. The standard (and auxiliary specifications like eg a graphics language) then describe operations that are act as release or acquire. For example various types of lock or mutex will be described as acquire and/or release, and if you think through how these locks are usually used, you will see why this makes sense. Then, below the language abstraction, the actual mechanism to enforce the required ordering is dependent on the hardware. Again as we have discussed, the roughest level is simply to flush various large buffers at release/acquire barriers; fancier

is to use “color” markings to allow some earlier or later memory transactions to slide past a barrier while others are not allowed to slide past, in a way that matches the required imposed ordering; fanciest of all (when you can achieve it) is strand ordering, where you know the subset of operations that’s relevant to the barrier and can allow everything not relevant to the barrier to do whatever it likes.

Orthogonal to all this is implementation in hardware, where the details are all about cache protocols and how we can minimize the amount of state we need to store in each cache, and transfer between caches, while still communicating what needs to be communicated; for example if core B reads a value that I have written to by core A, and which is in core A’s L1, then we want core B to see the value in core A’s L1.

Now just a little thought will show you that this is a different way of thinking from acquire/release, in a sense more primitive and less formal. Acquire/release thinking would say that if the address in question is a data address not a synchronization address, there’s not a problem (either your code has a bug, which is your problem; or the code doesn’t actually care about what core A wrote vs what core B reads). Either way, we should have that most values we write to cache A are irrelevant to what core B ever wants to see; and the values we do write should be marked in the code via acquire/release which will then (somehow) communicate the correct transactions to the hardware.

However the world is what it is. Essentially what we have right now is at the language level we can (and increasingly try to) be precise about which memory transactions and memory addresses are relevant to sharing and synchronization and which are not; the precise ordering is communicated via somewhat precise barrier instructions into the core; but then at the lowest level we use cache protocols like MESI that are, perhaps, heavier-weight than we really need, and which ensure that every core more or less sees what every other core is doing. Suppose, for example, that I, a non-threaded app, allocate a page of (ultimately physical) address space and execute some temporary storage there. Once I am done, in a perfect world, all the relevant storage (meaning various cache lines in L1, L2 and L3) would all just disappear. But in the real world of current MESI there is no way to indicate that; instead what will actually happen is that at some point the OS (likely running on a different core) will want to reuse the physical page; will begin by zeroing the page (so that the new user cannot see my old, possibly sooper-sekrit, data), and each zeroing operation will act as a cache-to-cache transfer through MESI, as snoops, cache line transitions, and lines of data are moved around.

There are tweaks to MESI to make it more performant (eg L1 to L1 transfer mechanisms, or perhaps a message to remote zero a line rather than transferring the line to my cache to be zero’d) but these are band-aids; ultimately we have an impedance mismatch between how MESI and caches grew up, vs how we formally want to use them.

BUT GPUs give a chance to do everything all over again! It’s only recently that GPUs acquired caches, and there’s still no expectation that their caches are “transparent” in the way that CPU caches are; the expectation is that you simply can’t get many types of synchronization you may want, and the synchronizations you can get you have to be explicit about.

This in turn allows for the possibility of very different cache optimizations (for example the “ephemeral data, never to be synchronized”) example I gave in the previous paragraph. Even if we decide that there is value in GPU L1 to L1 coherency, how should we implement it?

This is the background to a paper like (2015) <http://rsim.cs.illinois.edu/Pubs/15-MICRO-scopes.pdf> *Efficient GPU Synchronization without Scopes: Saying No to Complex Consistency Models*. You may think this paper is too technical, but it's worth persevering through it because it covers a lot. To avoid getting confused, realize that three different types of issues are being discussed:

- how do GPU's work today? Everyone agrees that between "kernels" there's an implicit release/acquire operation, in other words changes to global memory (many of them still in local L1D) need to be moved to somewhere where they will be visible to the rest of the GPU, probably to L2. So

- + between "kernels" there needs to be a flush of the L1 out to L2 and

- + we can't start a dependent kernel (however that's defined, but specifically one that might care about these changed values) until this flush is complete

- + you can see that this in principle already allows for some optimizations (eg: can we tag values in L1 by kernel, so that we only flush lines owned by the kernel that just ended, not all kernels? can we know that a subsequent kernel does not need to wait for the flush?)

- + I've put quotes around the word "kernel" because it's a matter of convention that we provide this implicit release/acquire between "kernels". If we know that we don't need this (internal graphics passes...) or we can explicitly define certain types of kernels as not behaving in this way (eg define a "Tile" shader that has specific and different rules, eg that it can only modify data in its tile, no changes to global address space) then we can skip this heavyweight transition between kernels.

- how could future GPUs work? For example could we define a light-weight version of MESI that operates between the L1Ds of the GPU? Or is that a foolish idea, we should stick with explicit software handling of such coherence as is required?

- suppose we did implement such a coherence protocol. Should we extend it to the CPU? In other words how much transparent coherency should we have between the CPUs and either the GPU L2 or L1Ds?

I'm not going to discuss this in any more detail because, once you appreciate what the issues are, there's nothing more to say. Apple and nVidia will presumably make their decisions, at which point we can discuss things further. But for now the situation is what it is.

- No hardware coherence in L1D;
- implicit release/acquire across some kernels (right now basically always for compute kernels, maybe omitted for some graphics kernels...);
- some mechanisms within Metal for enforcing barriers and/or memory ordering at very coarse granularity, with the understanding that these mechanisms can and might change between devices.

(In other words if you want to share data between the CPU and GPU, you are supposed to use an MTLSharedEvent to ensure synchronization between the two.

The MTLSharedEvent will ensure that the CPU and GPU each wait for the other to complete its task, as necessary BUT it will also impose release/acquire semantics on memory so that it will do whatever is necessary to ensure that GPU and CPU see mutually coherent memory. This could be

implement by having the GPU L2 be coherent with the rest of the L2 caches; it could be imposed by having the SLC know all the lines and their states in GPU L2 and doing whatever is necessary to ensure that lines move between CPU L2s and the GPU L2; or it could be imposed by flushing the GPU L2 out to SLC. You may think you know enough about the timing of the CPU relative to the GPU that you know the GPU will be finished by the time the CPU starts reading GPU data, so you don't need an MTLSharedEvent. What you are missing is that the MTLSharedEvent is not just performing timing synchronization; it is also performing release/acquire synchronization, in other words ensuring that, regardless of the details of the GPU L1D and L2 caches, what's in those caches will be made visible to the CPU by the time the MTLSharedEvent waitForEvent() completes – but may not be visible if you don't use an MTLSharedEvent.)

Apple Implementation

Introduction

Let's start with the patent (2013) <https://patents.google.com/patent/US9508112B2> *Multi-threaded GPU pipeline*.

The main value of this patent is that it confirms our understanding of the initial design point. It describes a barrel processor, making use of an operand cache.

Conceptually, at any given time, four threads are “active”. These threads run in lockstep (always 0, 1, 2, 3, 0, 1, 2, 3, ...) ie a barrel-processor design. This scheme

- basically makes the pipeline (to a specific thread) look like it is running at $f/4$ and so allows many low-latency instructions to execute back to back.
- non-variable-length latencies longer than this can be handled by compiler scheduling
- the most common minor hazards (bank collision, too many input operands, low bandwidth execution units) can be handled by static scheduling, and are simplified because we know that in any one cycle only one thread is allowed to perform certain read or write operations
- thus we should be able to run reasonable lengths of code with the same 4 active threads. Any remaining minor hazard (generally operand load) will result in the pipeline for that thread being frozen for one cycle while the other pipelines keep going. (You can envision this as, for this cycle, the frozen pipeline produces no output, so that its eventual output will be available next time for bypass).
- when a thread hits a major hazard (miss to L2 or DRAM), then scheduling moves it out of the set of four active threads and rotates in a different (no-longer-waiting) thread.

Going forward we'll see every element of this change.

Scheduling - what are the issues?

GPU scheduling is multilevel in the sense that there's a range from the highest level of scheduling (OS decides, on the CPU, what processes, with associated GPU work, to run next) through kernel scheduling

and threadblock scheduling all the way down to the most local scheduling (some queue decides which warp gets executed in the next cycle).

At each level of scheduling we have logical constraints (certain tasks cannot happen until earlier tasks are complete) and resource constraints; and we want to optimize performance while honoring QoS. All of these are large topics that we need to understand at least approximately to appreciate the entire system. We need to understand why logical constraints exist and how they are implemented, what resource constraints exist at every level, and how the GPU thinks of QoS.

This means we have to jump around somewhat in our presentation for what might seem most logical in some sense.

threadblock scheduling - optimize for local memory reuse

It would seem to make the most sense to start at the coarsest scheduling level, with kernel scheduling. But let's actually start somewhere in the middle, with one of the earliest (and thus simplest) patents: (2014) <https://patents.google.com/patent/US20160055610A1> *Gpu task scheduling*

This patent is specifically relevant to graphics (about which I know little) and is surely out of date. Meaning there will be doubtless be many inaccuracies and some outright falsities in my explanation, but the big picture should be correct, and I don't want to interrupt the flow by constantly saying "I think" or "it seems to be"...

So let's assume we're a GPU executing a graphics pipeline. What does that look like?

We can at least approximately map the stages of the graphics pipeline onto successively executed kernels, so these might look something like (simplifying dramatically) a Transform and Lighting kernel, then let's say a Rasterization stage that maps each polygon into fragments (lines along the x-dimension of the screen such that the set of y-offsets of all these fragments correspond to the visible polygon), then a Shading stage that textures each fragment.

A basic design (like nVidia appears to be, at least from the outside) would execute the first kernel (T&L) completely, one threadblock at a time, then the next kernel, then the final kernel, threadblocks handed out to cores based on the most-room principle. Note that, while simple, this scheme makes no attempt at exploiting data locality beyond dumb luck.

Consider in contrast how Apple (and for that matter PowerVR) might handle it.

Essentially we'd now have two initial kernels which (behind the scenes, this is not an MSL keyword!) might be called *Frame* kernels, which operate on the whole frame. The first executes a sequence of threadblocks to perform T&L, the second sorts the results vertices and polygons into tiles. These, more or less, have to follow the standard rules for kernels, like the second only begins once the first is completely finished, and there is no explicit data transfer between them, the first kernel flushes L1 caches before the second begins.

But then we get two new kernels which me might call *Tile* kernels which want to operate by a different set of rules. Now, once we start working on a tile, we want to, as much as possible, reuse tile storage from one Tile kernel to the next (ie we're no longer following the standard rules of flushing data,

whether it's cache data or Scratchpad/Tile data between nominally independent invocations). At an approximate level, the Rasterizer is doing something like, for each triangle that overlaps this particular tile, (as binned by the earlier stage) a fragment corresponds to a warp (if the widest a tile can be is 32, then no more than one warp), and a threadblock good enough corresponds to a triangle. Remember that we concluded that, for Compute the largest possible threadblock might be slightly larger than a core, midsized (fewer registers) threadblocks might fit two or three to a core, and even smaller threadblocks (not many registers, fewer threads, less Scratchpad) might be more than three to a core. For these triangle threadblocks, recall that a Tile is something like 32×32 in size (so 1K bytes, so let's say 8K storage if four channels, each of 16b HDR data), and most triangles are small, so many triangles (threadblocks), and many Tiles will fit the resources (Tile storage, registers) of a core.

If we want to maximize data reuse, we now want to do something like: do as much work as possible within a single Tile before we move on to another Tile (which may mean multiple successive steps of what look somewhat like different successive kernels, without flushing data between them).

This looks like a very different scheduling paradigm from nVidia, based on

- substantial “kernel” interleaving rather than running successive kernels
- scheduling that wants to be exploit data reuse.

In slightly more detail, then, each core is capable of supporting some number of tiles and these tiles are delegated to each core. (The patent doesn't talk about this, but the sensible thing would be to have neighboring tiles given to the same core, for obvious locality reasons.)

Each core is then given a stream of threadblocks (ie triangles) consisting of warps (ie fragments) to be shaded (ie colorized, so think eg texture lookup) which are buffered in some way. From this pool of threadblocks then, how do we schedule the next threadblock to execute? According to

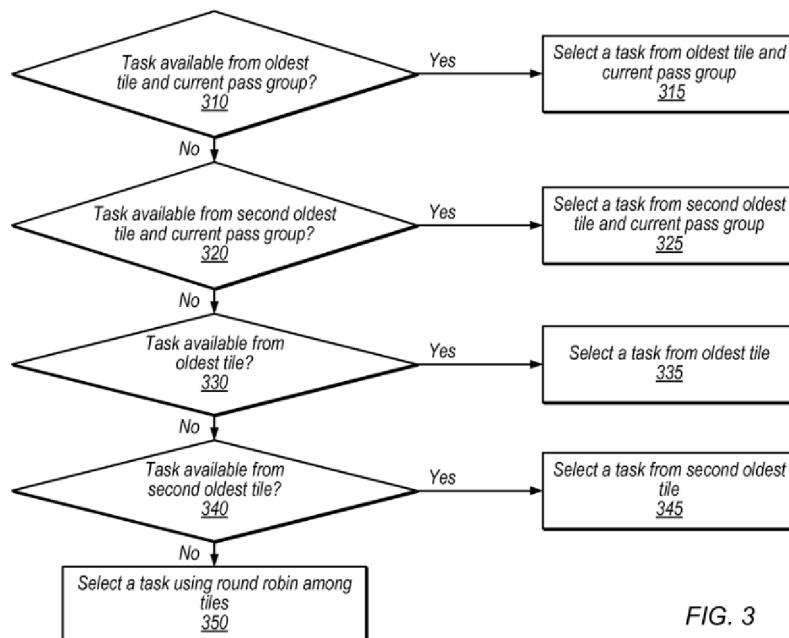


FIG. 3

Note the consequences of this.

We try to do as much work as possible within the oldest and second oldest tiles on the core. Maximal reuse of those tile's storage, with a backup plan if the preferred tile doesn't have a threadblock available right now.

We try to do as much work as possible within the current "pass" which will reuse storage that might be used across fragments (ie warps) ie this pass may be handling textures, and successive triangles within this Tile may use textures nearby in the same texture atlas, so whatever texture resources are brought in for this pass of this tile are likely to be reused in Texture, L1D or L2 cache. Likewise if the pass is a shadow map pass or whatever.

If we can't make optimal reuse (reuse Tile storage and reuse "Texture"-type storage) because no threadblocks are available, then try to make partial reuse (just Tile storage), likely moving on to the next pass. If even that is no possible, we've likely finished all the triangles of this Tile and we move on to a new Tile.

As I say, the details surely have inaccuracies and have changed; but it's already clear that this is a much more sophisticated scheduler that is trying to optimize

- as much 'kernel" interleaving as ordering rules allow for
- as much data reuse as API and internal conventions around Tile storage allow for
- taking into account information (like Tile relative age) that simply isn't recorded anywhere in some other schedulers
- while delegating work downward to some extent (there's a coarse centralized delegation of Tiles to cores, but then scheduling of Threadblocks happens at the core level, not at a central level).

All this presumably informs and (ideally can be utilized by) the scheduling of Compute kernels.

The reason this is especially interesting is that if you look across a large number of GPU compute, apps you'll find it's remarkably common to have most of the data in a subsequent kernel produced earlier by what is "geometrically" the same threadblock in the predecessor kernel, so essentially the same situation as we're describing above with data reused in a Tile across shading "kernels".

It certainly seems like we need some language level tools to indicate when two kernels can be "interleaved", so as to flow data from one Threadblock directly into the next "instance" of that same Threadblock in the successor kernel; and generally more explicit ways to indicate both dependencies (and their types) between kernels, and the extent to which we do or don't want to force cache flushing between kernels. Right now what we have is a set of conventions that grew up over time as hardware evolved, and that are neither optimal for hardware design nor optimal for ease of development or performance!

kernel scheduling - optimize for simultaneous use of different hardware

On the other side of things we have very coarse grained scheduling, at the level of what GPU processes or grids, or large batches of workgroups from a grid, to execute next. I don't know what the standards for this are or were, but looking at the patents, it seems like this was originally a driver task performed

on the CPU, and Apple has been trying to move as much of it as possible onto the GPU, as well as trying to make it more efficient.

As we have seen in our survey of nVidia, baseline kernel scheduling is remarkably primitive: basically execute kernels one after the other in first-come first-serve ordering. What counts as “sophisticated” improvements to this include:

- allowing more than one process to submit kernels
- tracking kernel dependencies, and starting a non-dependent kernel once the current kernel can no longer fully-occupy the machine, so that the second can use up threadblock slots no longer used by the first kernel
- a rough priority system that might move a high priority kernel ahead of an earlier enqueued kernel (but only affects the kernel’s position before scheduling).

Apple approach this rather more like CPU scheduling.

From the start there’s a presumption that we are dealing with processes (ie kernel’s exist in the context of a process) and process heuristics and explicit QoS settings are used to derive kernel priorities.

There’s also an almost inverted sense of how to optimize the GPU. nVidia (at least in the past, and if we ignore MIG) seems to prioritize running a single kernel at a time, with multiple kernels only kicking in under unusual conditions, like small kernels on a large GPU. Apple, conversely, expects to optimize performance by careful juggling of multiple different kernels, even on their small GPUs.

This leads to a very different type of kernel scheduler. If you want to optimize by executing multiple simultaneous kernels, you need to know what resources each kernel is going to dominate, so that you can pack “compatible” kernels together.

This is something of a knapsack problem, in that a particular workgroup requires certain amounts of resources (so many registers, so much local address space, maybe a particular hardware unit like vertex or fragment) and ideally you want to pack as much work as possible onto the GPU without the combination of tasks ever exceeding any particular hardware limit. So, eg, you want to pack together a kernel that is compute-dominated together with a kernel that is memory-movement dominated. Or one that is mostly calculating vertex geometry with one that is mostly using the texture units.

The most basic way to attempt this might be to have the compiler summarize, as best it can, what resources it expects a particular kernel to require, and this is certainly better than nothing.

But better would be to have live tracking of how different resources in each GPU core are being used, and making scheduling decisions based on that feedback.

The patent (2017) <https://patents.google.com/patent/US20180349146A1> *GPU Resource Tracking*. is essentially about having such a feedback system, and what you can do once you have it. It makes the following additional points:

- at least one aspect of how Apple handles QoS is not simply by moving one kernel ahead of another (something that becomes unclear when you are deliberately trying to run multiple kernels simultaneously!) Instead kernels are nominally allocated a fraction of the GPU, say 20% of the GPU vs 80% for

another kernel. Naively, then, this can be translated into submitting one threadblock from this kernels along with four threadblocks from another kernel.

- but this fails on at least three grounds.

- One is that kernel A's threadblocks might execute for much longer than kernel B's, so threadblock count is not a very good measure of "fraction of the GPU".

- The second issue is that the GPU provides discrete resources (for example Scratchpad capacity) and it may not be possible to pack one unit of kernel A and four units of kernel B simultaneously on a single GPU.

- The third issue is that threadblock scheduling is not the only appropriate level. Threadblock scheduling is appropriate for deciding how to split Scratchpad storage across kernels, but when deciding how to split registers across kernels, the appropriate level is each time a threadblock completes a warp and we have to decide which of the currently executing threadblocks will supply the next warp.

 - thus we

 - + track GPU occupancy more directly, rather than the easier tracking threadblock count AND

 - + we may toggle the GPU packing so that rather than, say, trying for a 20:80 split for kernel A vs kernel B, we may allow a 40:60 split for some period of time (say a millisecond) then for the next millisecond we switch to a 0:100 split, so that one average we still have a 20:80 split, but in such a way as to match the hardware resources.

On a latency compute engine (think CPU) QoS is mainly about which piece of code gets to execute first and thus finish first.

But on a throughput engine (think GPU) "finishing" is often an uninteresting concept; we execute code frame after frame, and a more important element of QoS may be sharing the *throughput* that the GPU is capable of delivering. So we might give a high priority kernel A 80% of the GPU and a low priority kernel B 20%.

(The way Apple actually seems to do this, judging by a large number of patents, is something like we start by giving a high priority task as many resources as possible, so like a latency engine, to allow it to finish ASAP. But if it doesn't finish right within a short time, then we seem to transition to more of a throughput sharing model where we give the high priority, and all the other, kernels a target fraction of the GPU.

The various QoS elements then juggle the "effective" priority of the task, determining eg which kernel gets arbitrated to launch the next threadblock, in such a way that, long term, each kernel hits the fraction of the GPU that it's allocated.)

This is all, however, far from the end of the story, because what exactly does "80% of the GPU" mean?

One answer is to give kernel A 80% of the GPU cores. But this is sub-optimal because one of the ways we boost our GPU throughput is by having different pieces of hardware executing simultaneously: both vertex processing and texture processing at the same time, working on behalf of different kernels.

A second answer is to give kernel A 80% of the time (say 80 microseconds) of execution, followed by kernel B running for 20 microseconds. But this sequential execution also does not use the same hardware in parallel.

What we want is something like execute both kernel A and kernel B simultaneously on the same core, but with kernel A given 80% of “the resources” of the core, and kernel B given 20%. But even this needs to be clarified.

Supposed for example that we have 100 registers, and both A and B require 50 registers to execute.

A naive implementation of the above idea might implement something like a static resource allocation: A gets 80 registers, B gets 20, and A gets allocated warp after warp because there are never enough registers to launch a B warp. This is obviously both unfair to B and dumb (since we might as well allocate two A warps rather than reserving 20 for a B that can never launch). This clearly looks dumb in this example, but it won’t obviously look quite so dumb with real numbers – multiple warps possibly executable. It does, however, show the limitations of a *static* attempt to impose QoS by partitioning core resources.

Much better is that we do something like the following: we start by executing two A warps, then an A warp and a B warp, then back to two A warps, etc. Now we have essentially a 75:25% split, not quite 80:20, but close, and we’re using all resources. The model is more or less

- start with a desired fractional utilization and a static resource allocation; and schedule warps based on what fits into that resource allocation and “effective priorities” (which translate into something like weighted round robin)
- every so often compare the desired utilizations to the actual utilizations and in response change the static resource allocation and the effective priorities

Ideally between the two of these we converge on a resource allocation that shares resources well, *and* a warp allocation that matches the throughput allocation we have in mind.

To do this requires

- machinery that’s tracking resource utilization over some period of time (warp by warp, or microsecond by microsecond)
- machinery that is accumulating these resource values over longer periods of time, and over all cores not just one core, to see how resources have been split, on average, between kernels
- machinery that conveys these averages and how they differ from the desired averages to the core, to adjust warp by warp, and threadblock, scheduling.

This scheme is *dynamic*, in the sense that, even though the currently executing kernels have been decided and threadblocks from those kernels are being handed out to each core (in accordance with the above ideas), it’s possible at any point to override the threadblock allocations. If a new much higher priority kernel arrives, it will move to the head of the queue and have its threadblocks handed out next until it is complete. This is not “complete” preemption in that it relies on the enqueued threadblocks completing in a timely fashion, but it does give some degree of allowing say a split between

- visible on the screen work
 - computation that will affect the screen (think game effects)
 - background computation that is not visible
- and ensuring that mostly what the user sees never glitches, even as the GPU is allowed to perform

background work (eg photo classification or whatever)

The patent also suggests (without giving details) that the scheme is even more sophisticated, in that it takes rates into account, so that processes which are high priority, but also known to generate work at a fairly low rate may sometimes not get a slot you'd expect, simply because that slot is a good match to some other process, and from the recent historical rates we know that soon enough there'll be a better slot for this high priority work.

One final point worth bearing in mind is that it is probably the case that some of this scheduling, and much of the GPU “outer level” work (down to the decision of which batches of threadblocks to hand out) is done by firmware on the GPU companion core. This means that some changes can be made even after the chip ships, and a clever tweak to something like the kernel scheduling algorithm may result in a noticeable improvement. I personally noticed that an OS update to my M2 Pro resulted in h.265 HW encoding becoming noticeably faster, probably as the result of a similar change to how work is scheduled by the Media Encoder’s companion core firmware.

The focus of the patent is the scheme for monitoring the use of various GPU hardware. Once you have such a scheme, how else can you use it? The second use the patent suggests is that you can use it to tell when a GPU core appears to be locked up, and can then take action more quickly, and at a finer granularity [at the individual core level] than traditional schemes for trying to detect a hung GPU.

(2017) <https://patents.google.com/patent/US20190042312A1> *Hardware resource allocation system* fills out (very roughly) something of how we use the resource utilization machinery earlier described.

To summarize the point of the patent, the simplest resource allocation system (imagine eg we are handing out blocks of registers) simply maintains a list of free vs allocated resources, and looks in that free list when performing an allocation. One consequence of this is that the only way resources can be freed is when the owner asks for them to be freed, because there is no other centralized record of ownership.

The patent describes augmenting such a free list with ownership records, which means that when allocation is requested and resources are not available, the centralized resource allocator can see if there is some appropriate victim thread from whom the resources could be taken. (Once this decision is made, there are many ways to proceed which the patent does not cover. If the only limited resources are registers, we could, eg, wait for a low-priority warp using lots of registers to complete. Or we could pause the warp and swap its registers to cache. If the limited resources are both Scratchpad *and* registers, we might conclude that right now resource allocation is hopeless, the requester needs to give up for now and try again when this threadblock is complete.)

The main thing is we can make informed decisions rather than always simply giving up immediately if not enough resources are available.)

That's the summary; below we discuss the issue in much more detail.

So far we have described (or rather implied) a *kernel* scheduler that consists mainly of lists of kernels together with their dependencies. For each of the pool of kernels that could be executed (ie are not waiting on an earlier kernel dependency) we have some associated resource information, perhaps pre-supplied, perhaps measured based on some already executed threadblocks; and some other information (eg QoS, age). From this pool we hand out tasks (ie threadblocks).

We have described this task in terms of trying to minimize competition for short-lived resource usages, like texture, compute, or load/store units; and conflicts over the use of such units are resolved at the lowest level of scheduling, deciding clock-by-clock which warp to execute next from some pool of active warps.

But there are other “hardware resources” that are longer lived.

Conceptually, for example, Scratchpad memory is allocated for the lifetime of a threadblock; while registers persist for the lifetime of a warp.

For example one can imagine a stage in the graphics pipeline that consists of multiple tasks each walking a queue of triangles and handing those triangles over to a rasterizer that converts the triangle into a list of scan lines, each scan line then to be handled as a single warp and filled in by a pixel shader. On the one hand, once the rasterizer has been allocated to a task, it is fully occupied handling triangles from that task as long as such polygons are available, ie it is “reserved” for that task; on the other hand that task may freeze up and be unable to provide polygons every time it encounters a memory miss and has to load more triangle data.

So think about this at the abstract level. In the simplest model, the scheduler simply hands over a threadblock to a core and it’s the core’s responsibility to allocate the resources for the threadblock, whether those are Scratchpad address space or registers for a warp.

In this model, if *all* the resources cannot be allocated, the new threadblock simply sits around in some buffer until enough currently executing items (warps, or even full threadblocks) have completed and freed resources to the extent that the new threadblock can begin.

This is clearly sub-optimal in that in a perfect world that core might have been given a different threadblock, one able to immediately utilize the resources available. The scheduler described in the previous section “balances” workloads at the level of short-lived execution units, but not at the level of longer-lived “reserved” hardware resources.

To put it even more bluntly, in a perfect world we would do things like balance a threadblock that needs lots of Scratchpad but not many registers against a threadblock that needs lots of registers but not much Scratchpad. But this is a “split-level” decision – it requires taking into account Scratchpad (relevant to when threadblocks are scheduled) and register usage (relevant to when warps are scheduled).

So how could we do better? Again the key is to provide the decision maker (ie the scheduler handing out resources) with more data.

We start with a centralized unit responsible for allocating these sorts of longer lived resources, and this unit tracks how many of each resource have been allocated on a core. This is already a good start, since we can now query this information to know which of the possible threadblocks we have available to

hand out could immediately start running on the core.

But we can do even better. The centralized hardware also tracks the state of the owner of a particular hardware resource and if that state is *inactive* then we may be able to transfer state ownership to the new requester. Inactive may mean waiting on a memory load, or may mean waiting on a synchronization event; something that is expected to last a few hundred cycles or so.

The patent is more than a little opaque on exactly what this means, but we can imagine a few options. For example suppose a task that has reserved a fragmentation engine, as described above, is waiting on memory for some more triangles. Then we can take away its fragmentation engine and give it to a different task (presumably also marking the first task in some way so that it can get its reservation back when it is ready to become active again!)

You could even imagine something like this at the register level: eg hand out registers in blocks of 16, copy a block of 16 registers of an inactive warp to the L1D, and give those 16 registers to a warp of the threadblock that has just been allocated to this core.

As I said, the patent is maddeningly opaque with no actual examples ever given! But the overall idea seems to be to try to get value out of “hardware resources” that would otherwise be idle when the current owner of the resource is blocked/inactive.

Finally we have spoken above in terms of a single resource allocation. In actual fact, to begin the execution of say a new threadblock, perhaps three different resources (of varying sizes) might be needed. A naive allocator might first try to satisfy resource A, then resource B, then resource C, in the process affecting three different inactive tasks. A smarter allocator would look at the entire situation and realize that by removing resources from say just one of these tasks, it can satisfy all the requested resources, while the other two tasks can continue execution as soon as they are reactivated on seeing their load data, or whatever. (That way we will keep the machine more fully occupied all the time, because these two tasks will remain active on the core as possibilities for clock-by-clock scheduling; we won’t have to wait until they can re-acquire the resource taken from them).

And, just like the previous patent, once you add this sort of tracking machinery, you can also use it in other ways. Specifically the patent suggests that since we have a centralized controller that knows the owner and usage stats of all hardware resources we can also, if appropriate, set a cap to the resource usage of particular kernels or processes, as another way of enforcing elements of QoS.

There are some overlaps between this patent and the previously discussed *GPU Resource Tracking* patent. On the CPU side we repeatedly (eg for power control, or bandwidth control) saw a pattern of implementing hardware to track and control a resource in a somewhat distributed fashion, followed by building on that resource in a centralized optimal fashion. The same pattern seems to have played out with resource allocation. These 2017 designs lay the groundwork in terms of tracking what is happening (who owns which resources, and how actively are they using them), but split across multiple different resources, sometimes handled in different ways. We now evolve this to a unified allocation and control scheme that can do all of the above, but also a whole lot more.

resource allocation (2017 and now)

We've now seen that resource allocation is an important part of scheduling, not least in the sense that we cannot start the execution of a new task (whether threadblock or warp) until certain hardware is allocated.

Consider now the allocation of "memory-like" resources, eg Scratchpad storage for a new threadblock, or registers for a new warp.

How might we do this?

The simplest answer is we don't really try. One threadblock gets the entire scratchpad, each warp gets a hardwired set of registers, no attempt to do better. This may sound dumb but it is, for example, right now the way AMX handles registers, with a dedicated register set for each potential client core in a cluster...

If we want to do better, the next step up is to provide each active threadblock with a base and a length register. The first threadblock sets the base to 0 and the length to whatever was allocated. The next threadblock sets base to previous length+1, and so on. We keep going until we run out of Scratchpad storage. Each access to Scratchpad adds the nominal address to the base, while simultaneously testing that it is smaller than length, and we're set. Once an allocation fails, we wait until an earlier threadblock complete and releases its range, then try again. There are a variety of ways we can improve on this basic idea (read any discussion of malloc!) but what's worth doing depends on the details – what makes sense if you expect hundreds of thousands of simultaneous allocations is different from if you expect up to maybe eight simultaneous allocations.

This scheme as described is not utterly terrible (and readers may recognize that variants of this were used in early mainframes, and in the very earliest PC operating systems).

If you want to make this work better you might want to add a few elements.

Firstly you might want to aggregate units together. Allocate Scratchpad maybe at the granularity of a "cache line" (64B or 128B?) or even larger like 1KB. Allocate warp registers at a granularity of maybe 16. Second you will want a bitmap or something similar to track the free vs in-use granules. And hardware that can rapidly find runs of bits of a particular length, so that you can easily find a run of three (or whatever) bits set to 1, to match a three granule request.

At this point you have an allocator adequate for small purposes, and this is essentially the content of (2017) <https://patents.google.com/patent/US10698687B1> *Pipelined resource allocation using size aligned allocation*.

The specific detail being patented is how, after a few rounds of allocation and deallocation have been performed, do you choose where to perform the next allocation? For example options you might imagine include

- allocate from the smallest free block that meets the requirements
- allocate from the largest free block that meets the requirements

- allocate from the “oldest” (in some sense) or “youngest” free block that meets the requirements
- allocate round-robin style (ie just keep moving left till you find a block that’s big enough)

What Apple does is none of these, but should seem familiar from some malloc designs. Allocate blocks by “alignment” granularity.

So if we want to allocate a block that is say 5 units in size, we will look for runs of 5 consecutive free blocks, at locations 0, 5, 10, 15, ...

If we find more than one such location, we will choose the smallest fit.

This behaves somewhat like a malloc that keeps separate free lists for blocks of different sizes; while not requiring the overhead of such a design.

This scheme is implemented in hardware, and performs the allocation (or reports a failure) in one cycle. Not bad!

This is far from the end of the line, however! Suppose you want to do better. The big problem with the design above is that we get fragmentation. We may have three granules free, but not contiguous.

The first step to a more powerful design is you introduce a level of indirection. Add something like a TLB that maps a virtual granuleID to a physical granuleID. Now we no longer need to care about contiguity; a three granule request merely needs to find three free bits anywhere in the bitmap, and set the appropriate (sequential) mappings from the virtualID to the (non-sequential) physicalID. And every granule access goes through the TLB lookup. This is, of course, basic VM and should be very familiar. To avoid confusion we will call this “TLB” an rTLB (resource TLB).

The next step is to also allow oversubscription. Now we no longer even require that there be three free physical granules, we simply hand out three virtualIDs, all mapped to “invalid” in the rTLB. To make this work, we also need

- a faulting mechanism, so that when access is made to an “invalid” entry, some sort of hardware kicks in to perform a physical allocation
- a means to handle oversubscription, ie what to do if there is no physical allocation available?

For memory-like resources, both of these are fairly easy in the sense that you simply need to copy out some memory to some alternative storage, and reallocate the block that was copied out. Obviously you know how this works for standard virtual memory, and we can do the same thing here. For example if the resource of interest is Scratchpad, we can copy a granule of Scratchpad storage out to the L1D, reallocate that granule as invalid in the mapping for the current user, and set up as valid for the new user, and then keep going.

Just like with VM, you can then use various heuristics to ensure that this swapping does not get out of control.

For a GPU two obvious such heuristics are

- we already know which tasks may be inactive waiting on either RAM or a synchronization event, and obviously it makes more sense to reallocate from them than from an active task

- we also expect that, generally, warps and threadblocks will tend to behave in a similar fashion.

So we can do something like

- + for the first round of warps of a threadblock, begin them with just one of their register granule

allocations hooked up to a physical allocation and see what happens. If no faults are generated, we are fine! If faults are generated, then for the next round of warps, try to ensure that two register granules (or however many are required) are hooked up before starting the warps.

Of course the details of exactly how aggressively you oversubscribe will depend on the cost of re-allocation faults.

This next technology stage was introduced with the A17/M3 GPU and is given the brand name *Dynamic Caching* by Apple. It is especially valuable for “unpredictable” shaders. For example a shader may have one common (data dependent) path that only requires a small number of registers or Scratchpad, and a different rare (data dependent) path. The shader has to allocate the full possible set of registers or Scratchpad storage it might require, but usually will not use most of its allocation. By only allocating resources at the point they are truly required, we can frequently pack more threadblocks and warps onto each core, meaning fewer dead cycles when the core simply cannot find a warp to schedule.

The fairly recent patent (2024) <https://patents.google.com/patent/US20250068564A1> *Graphics Processor Cache for Data from Multiple Memory Spaces* describes some details of how this all works. I’d recommend at this point you download Volume 8 (Recent updates since I wrote these initial PDFs) and search for “*Graphics Processor Cache for Data from Multiple Memory Spaces*” which will give you a more detailed description of how the cache works at both a logical and a physical level.

However Apple’s scheme is even more ambitious!

What I’ve described above is a scheme that’s basically a simple remap table per core, with the ability to fault on certain accesses.

Compare this to the virtual memory system of a modern SoC. Two differences are

- VM remapping is common across the entire SoC. There are multiple local remap tables (ie TLBs) but these are all caches of a single system wide collection of remappings that we might collectively call the “page tables”. The TLBs are local caches of this entire set of remappings.
- To deal with the size of all the remappings, the “page tables” have a hierarchical structure which lends itself to multiple levels of caching.

This same sort of machinery is duplicated across the GPU. Superficially it looks like the CPU page tables (because of course it’s designed using the same ideas) but it’s not a remapping of multiple process virtual address spaces into physical DRAM, rather it’s a remapping of multiple GPU-specific address spaces (eg the Register address space of each Warp, or the Scratchpad address space of each Thread-block) into a common address space which refers to L2 SRAM storage in the GPU.

It’s hard to be certain, but I think what’s going on is that

- within a GPU core and L1 all accesses are via what is essentially (ASID, usage-specific-address), for example (ASID=registers for SIMD 27, address=register for thread 15 of SIMD 27). This is translated via hash into a location in the L1 SRAM, as described in the *Cache for Data from Multiple Memory Spaces* patent.
- when an address leaves the L1 for the L2, it’s translated into the “virtual address space”, or something similar, for the relevant process that provided the kernels. Thus the addresses in the L2 are not in

physical space and have an ASID tag. Doing this would allow the relevant TLB and memory lookup from L1 to L2 to be optimized for the task, without requiring the complexity of the full ARM 64b physical address space MMU. (For example the L1 to L2 TLBs don't have to worry about TLB manipulation broadcasts sent out by the OS to handle address space remappings.)

- when an address leaves the L2 for the SLC it is mapped from the process virtual address space to the physical address space.

One could imagine a few different variants on the above idea, and I'm not sure quite which Apple adopts.

To put this slightly differently:

Apple define the target address space (the L2 address space) not as physical SRAM on the GPU but as the device-wide virtual address space for the relevant process. Thus, in principle, a register lookup translates the (warpID, register number) via the rTLB into an (ASID, virtual address).

This seems clumsy, like it would then have to be translated by a second TLB into a physical address when we go out to SLC.

But not really!

In fact the entire GPU operates purely on addressing based on (ASID, virtual address). The ASID is what we have earlier called a *context*, a small identifier. There is no virtual address look up inside the GPU; virtual addresses only map to physical addresses when they leave GPU L2 and have to enter the rest of the SoC. (This is in contrast to eg nVidia, which uses virtual addresses in L1, but converts virtual to physical when addresses leave L1, so L2 operates on physical addresses.)

If things are done this way, then the fact that the target of this common address space is this virtual address space means that, in principle, data can expand out past the local per-core SRAM (ie "L1 cache") to the GPU-wide L2 cache, and you can start to do things like nVidia has done, where you move beyond limited sized threadblocks to allow, in principle, things like a large threadblock crossing four cores, and using a large amount of Scratchpad, based on aggregating the SRAM available across four cores into a large common Scratchpad storage).

Elements of this scheme (virtualize resources, and handle oversubscription by faulting) are described in (2020) <https://patents.google.com/patent/US20210271606A1> *On-demand Memory Allocation*. The patent as described would appear to apply to literal memory spaces (obviously Scratchpad, and also the private address space used by Ray Tracing) but experimental investigations suggest that it is more abstract and applies to other resources, specifically it does appear to apply to warp registers.

In other words you should think of this as a scheme for virtualization, generically considered.

Once you have a scheme that looks like VM, you can in principle use it in ways like VM!

The patent suggests three possible extensions to how this might be used.

First suppose you need to move work from one GPU core to another. In principle you could have each task (at the appropriate granularity, warp, threadblock, kernel, whatever) have the taskID act as an rASID, and simply move the task from one core to another. The first access by the core would miss (rASID+address) in the new core's rTLB, so the rTLB would be filled in from the central rTLB. Then the

reference to the address (say a register) would miss in the new core's SRAM, and so the register data would be moved from SRAM in the previous core to SRAM in the new core. Very much like moving a process on a CPU. This seems to be what Apple ultimately envision, and you might want to do this for all the reasons you might want to move a process, from load-balancing across cores to consolidating work on a few cores and powering down the rest.

Second you can share "memory" between tasks. We can already share registers between the lanes of a warp in the form of uniforms, but the uniform has to be reinitialized for each warp. You could imagine something like a threadblock, or even an entire kernel, sets a block of registers to generally useful values, then that block gets "mapped into" every warp when the warp is activated. This both avoids the initialization instructions and allows the storage of these registers to be reused across each warp, rather than per-warp register allocation.

Something similar may be useful across threadblocks.

You could even do things like Copy-on-Write, though it's unclear how useful this might be.

Of course it will take changes to MSL and the compiler to make these ideas visible, but we can imagine them coming; the primary constraint on how powerful they might be is the extent to which "faulting" is a GPU-wide notification versus a core-wide notification.

Third once you have consolidate all storage in a single physical pool of SRAM, with this "resource page" indirection determining whether the storage is used by Scratchpad, Ray Tracing, Texture, Registers, or anything else, then you can even have the indirection route to the SRAM pool on a different core, as we have seen nVidia does for Scratchpad storage (though apparently nothing else). You can thus start to imagine things like threads to work together on a very large kernel across multiple GPUs, sharing Scratchpad storage. There are issues here in terms of enforcing L1 cache coherence, but we know how to solve coherence, and in the more constrained environment of a GPU cheaper solutions may be possible.

The patent as described by Apple is extremely ambitious, and it is likely that elements of this will continue to roll out over the next few years. Probably the version we have right now in the M3 as of 2023 is the simplest possible version (remapping and oversubscription within a single core, but not yet a common cross-core address space, moving address spaces, shared address spaces, etc).

A final technical point is that the patent talks (though vaguely and the details are unclear) about having multiple validity bits attached to elements in the page tables. This looks like a coalesced page scheme (a way to represent multiple successive pages with identical characteristics by one entry in a page table or a TLB entry). We shall discuss this in rather more detail later when we get to discussing the cache system and the "traditional" TLB.

We also have (2017) <https://patents.google.com/patent/US10699368B1> *Memory allocation techniques for graphics shader* which is uninteresting and content free, except insofar as it basically confirms various points we've already established:

- Apple wants to execute multiple different types of shaders on the GPU, including sometimes both

compute shaders and “pure graphics” rasterization

- this means there will be competition for use of the pool of joint storage that is used both for Tiles and for Scratchpad

- the system deals with this by dynamically throttling either the compute stream or the rasterization stream so that neither dominates the storage allocation (presumably using the QoS tradeoff mechanisms we have discussed so that each gets, on average, whatever fraction of the GPU it has been allocated).

Obviously this means that blocks of storage from this Scratchpad storage are being allocated and reallocated in some fashion, presumably according to the previous patent, but it’s unclear (my guess is not yet) if, at this stage, we are still subject to fragmentation or if we have already moved on to using indirection to avoid fragmentation.

One can get something of an idea of what these patents are about, from a different angle, by looking at (2016) https://people.inf.ethz.ch/omutlu/pub/zorua-holistic-GPU-virtualization_micro16.pdf *Zorua: A Holistic Approach to Resource Virtualization in GPUs*. This paper (referencing how GPUs more or less behaved in the years before when it was written) refers to how code is required to specify the sizes of various resource allocations (like the size of a threadblock, the number of registers used; and amount of local address space); and the dire performance consequences of getting these decisions wrong.

This seems to be the background to the Apple resource allocation patents. To us coming from CPUs, they seem kinda obvious, but for GPU’s dynamic resource allocation, sharing, swapping, and other virtualization techniques are currently leading edge ideas.

Something of a “wrapper” around both the previous work and perhaps some of the next work to be described is (2018) <https://patents.google.com/patent/US20190244323A1> *Pipelining and Concurrency Techniques for Groups of Graphics Processing Work*.

Consider the transition between one

block of work and the next. (I’m being vague in the terminology because the same idea can be used at multiple levels.) The most naive way to do things is eg, the wait till a workgroup is completely finished, write various data from L1 to L2 cache, synchronize or clear various registers, then start allocation and moving data for the next workgroup.

Better would be if these tasks could be overlapped so that clearing out of the previous workgroup (or higher unit) happened in parallel with ramp-up of the next workgroup.

But this overlap can’t be done if various special purpose registers (defining things like workgroup geometry or Local Address Space allocation or whatever) are required by both the exiting and the entering workgroup.

The solution is to duplicate the most critical of these registers so that the exiting workgroup can use the “old” set of registers while the entering workgroup uses the “new” set.

This patent seems to me a good idea, but it’s marked as abandoned. My guess is that legally it turned out to be indefensible in some way (to be fair, while a good idea, it’s also a very obvious extrapolation of existing ideas used everywhere), and that even though unpatented, it still reflects actual Apple design.

reminder – memory and coherency in GPUs

Before we go further, let’s again remind ourselves about memory issues, because these different from a CPU.

As Apple has made a big deal of, Apple Silicon provides a Unified Memory Model, but what exactly does that mean?

Part one is that there is a single shared pool of *physical* memory, not one physical VRAM and a second physical DRAM. This, plus the shared SLC, has the potential to mean much less explicit copying between the CPU and the GPU.

Part two is a shared *virtual* address space. If the CPU part of an app creates some data at a particular virtual address then, in principle, it can transfer that virtual address to the GPU and have the GPU read from that buffer, without either explicit copying or explicit translation. You can imagine some of the details required for this: both CPU and GPU need to use TLBs and common page tables, which in turn means things like the GPU has to know the processID associated with each block of code+data on which it wants to work.

BUT there is a third part that's required for this to work, namely "shared coherency" between the GPU and the CPU, and that's not quite what you expect. In particular the GPU is not 100% all the time, no questions asked, no limitations, internally coherent, or coherent with the CPU. This is to reduce power and area. Meaning there is some degree of thought required when handling shared data.

within CPUs

Backing up, if we have code running on two CPUs, one CPU writes to a particular physical address, then later the second CPU reads from that physical address, we expect that the second CPU will see the result written by the first CPU. There are some details here:

- how much "later" does the second CPU have to perform its work? We can't control the relative time of threads, so we probably need some sort of lock to ensure that CPU2 only performs the read AFTER CPU1 has performed the write.

But that is just about the timing; as long as the timing is OK, CPU2 will read what CPU1 wrote.

- what if CPU1 writes to two or more locations. Can CPU2 see the reads in a different order from the writes? Possibly yes, if the reads and writes happen close together, and that is the function of memory barriers, to ensure that a set of writes that act together are read as a *single transaction*, not split so that some of the writes are seen while others reads see stale data that has not yet been overwritten.

But again this is not exactly about CPU2 not seeing what CPU1 wrote; it's about the timing maybe being reordered from what you expected.

If you think about this, it's less obvious than expected that this works, and works well! CPU1's write is sitting in the L1D of CPU1. CPU2's read will look in the L1D of CPU2. You could have that the same cache line lived in both caches, so CPU2 could see the old data local to CPU2, not the new value written by CPU1. You could have that the request missed in CPU2's L1, went out to L2 and then to DRAM. How does the request know to be redirected sideways to the L1D of CPU1?

It is the technology of cache coherency (MESI, snooping, and all that) that makes this all work at tolerable costs in overhead and energy, but there are real costs here. And *most* of the time those costs are unnecessary, in the sense that almost every line accessed by a CPU will never be accessed by a different CPU; all the proto-

col messages, snooping, and cache state tracking are irrelevant to the behavior of most lines, but regardless that cost has to be paid. You could imagine, in principle, something like the system including blocks of memory, perhaps defined by address ranges, that made no promises about transparent coherency across caches; but CPU systems did not evolve that way.

within GPUs

GPU's evolved in a different direction that began with no caches on the devices, and a programming expectation of no coherence; in other words no promises that a write made to address X will be seen in a "subsequent" read from address X by a different "part" of the GPU, "part" defined very broadly.

You can see why this started. We have no idea as to the relative execution ordering of thread-blocks, or warps within threadblocks, so you can't really make assumptions (except within a single warp) as to the ordering of reads or writes to a particular address...

So when caches were added to GPUs, the approximate rules (never really formalized) governing them were essentially something like

- if kernel B is (somehow) indicated as dependent on kernel A, then we guarantee that the full set of writes to global address space by kernel A will be seen by kernel B
- if kernel B is not indicated as dependent on kernel A, then it shouldn't be reading from any global address space written to by kernel A, and who knows what it will read, no guarantees
- within kernel A or kernel B, beyond a single warp it is an error to require some sort of sequencing between reads and writes to a common address in global address space. Warps can co-ordinate (and use fences to synchronize ordering) within a threadgroup and in Scratchpad address space; but as a consequence threadgroups are limited in size and each entire threadgroup executes on a single core.

So those are the rough rules. How to implement them?

The general consensus has been that we give each GPU core an L1D (more or less the definition of a GPU core is a bunch of other hardware that shares a common L1D). This L1D makes no attempt to know what's happening in other L1D's; there's no MESI, no snooping, no state tracking.

Next we have a shared L2 visible to all the GPU cores.

Finally at "synchronization points", when a kernel ends, all the writes it performed are flushed from L1D to L2, and the L1D is invalidated (because even read-only lines within the L1D may have been changed by a different L1D, and may now be out of date relative to the L2).

This leads to some obvious (and non-obvious) ideas for improving performance, but it also raises the question of what the rules are for the L2. These have been less clear over time.

The L1D rules I described are kinda obviously necessary to get anything to work; but if you're, say, an nVidia card, a kernel has ended, it changed data at various addresses and those changes are now in the L2 and perhaps on the card VRAM. Do you owe anyone any sort of *automatic* synchronization of the L2 changes with the CPU?

In the past this didn't even make sense! The VRAM was expected to live in a completely different

address space from the CPU DRAM, and stuff only moved back and forth by means of explicit CUDA copy calls. Over time, the two address spaces became more and more common, till we now expect that they (kinda sorta) automatically stay in sync, handled by tracking write events to pages on both the CPU and GPU.

But there does not yet seem to be any sort of agreement as to what sort of *automatic* coherency you can expect between the L2 and the CPU; if you require the CPU to know of any changes made by a previous kernel, then you need to call an explicit very heavy weight fence that will (effectively, who knows of the exact implementation details) flush the entire L2 out to an L3 or SLC cache shared with the CPU.

(There are other ways you might do this.

- The CPU solution is that the GPU L2 is in constant communication with the SLC and the rest of the system, so that at any random time a read by a CPU core from a memory address that's currently "located" in the GPU L2 will return the most recent value. That's what we're trying to avoid.
- The simple solution I described is that after a burst of GPU compute, if for whatever reason we want the CPU to be able to see the results, we flush the GPU L2 out to SLC/DRAM.
- But an alternative is we don't actually move all the data out of L2 to SLC. Instead what we do is tell SLC the addresses of every modified cache line in L2. Later, if the CPU wants to access that line, the request will eventually reach SLC which will consult its directory and request the line from GPU L2. This is kinda like having MESI operate for the GPU L2, but not all the time, instead at a coarse granularity. If we expect that most of the time all the CPU wants from the GPU is a few bytes of summary result, not necessarily a whole lot of intermediate data, then this might overall be a better solution (more performant and lower energy.)

The rules for Apple/Metal essentially match this framework, with some tweaks to optimize the scheme for Apple specifics.

There is a basic unit of work called a *kick*. For a GPGPU developer, a kick is the same thing as a kernel; but more generally a kick can be some sort of fairly heavyweight internal graphics operation generated by the graphics specific hardware (eg some sort of large tessellation operation or whatever), hence the use of this more general term.

The *defining characteristic* of a kick is that it is a unit of work that does not assume it can see the results of writes by other warps of the kick. Thus, for example, the graphics system, with knowledge of exactly what's being done where, may bundle into a single kick two or three successive internal graphics operations that we might think of as multiple kernels, but which Apple knows do not need cross-warp memory visibility.

Dependent kicks must be separated by behind-the-scenes operations that flush the L1Ds on which the kick executed out to L2. Independent kicks can start executing right away.

At this point it should be no surprise that Apple tries to schedule things so that independent kicks are scheduled interleaved with dependent kicks, so that the flushing to L2 can happen in parallel with the execution of the new kick.

Apple also introduce a concept that's beyond traditional GPU's in the form of the *Tile Shader*.

Traditional kernels, as explained, execute as a single unit, with no sharing *inside* a kernel except within a single threadgroup; and with no sharing *between* kernels, hence L1 flushes between kernels.

But what if you have some data within Scratchpad/a Tile, and want to execute a sequence of transformations on that data?

In traditional GPU coding, this would require you to coalesce the transformations together into a single large shader that acts on a single threadblock.

In the graphics context, Apple allows you to write a different type of shader, a Tile Shader, that executes as part of a pipeline of transformations to generate a single image tile. This pipeline might begin with rasterization to the tile (Apple code), execute multiple Pixel Shaders (developer code), then execute multiple Tile Shaders (more developer code), constantly reusing the data in the Tile memory without a cache flush.

However, at least for now, Tile Shaders are part of the graphics world; there isn't yet in the pure Compute world something analogous to give us alternative ways to compose work localization beyond the construction of large monolithic kernels :-(

This concept and expectation of a sequence of related kernels that all operate on the same Tile of some underlying storage doesn't always make sense. It requires

- (a) that there be a reason to write code as a sequence of independent kernels rather than as one large kernel
- (b) that data used by successive kernels be localized to some storage like a Tile.

For many science use cases (a) does not seem to hold. You'd probably write a PDE or matrix manipulation code or suchlike as a single large kernel that calls functions, rather than as a sequence of somewhat decoupled kernels.

However for Machine Learning:

- the current pattern is to write Neural Networks as a sequence of independent kernels that flow data from one kernel to the next so (a) holds, and
- for at least some Neural Net layers (eg pooling or convolution, though not all types of layers) there is locality in the data that flows from one layer to the next.

So, especially for researchers experimenting with new types of neural networks, there would seem to be value today to allowing something like Tile Shaders to be visible to pure Compute outside graphics.

There is a higher order grouping of work called the Command Buffer which corresponds to a collection of kicks. The defining characteristic of the Command Buffer is that between two Command Buffers the L2 is brought into coherence with the rest of the system.

Thus (in simplistic terms)

- a number of kicks occur,
- after each one data is flushed from L1D to L2;
- then the Command Buffer ends, and
- we see flushing from L2 out to SLC.

If you look at the Metal Best Practices docs, they suggest using as few Command Buffers as you can possibly get away, ideally one per frame. Now you know why Command Buffers are considered expensive; more precisely it's the boundaries *between* Command Buffers that are expensive because they

correspond to an L2 flush.

This is what various patents seem to say, but the extent to which it's what *actually* happens on Apple Silicon is unclear!

The problem is we sometimes don't know if the background in a patent is trying to cover the *whole* of Metal (including, specifically, Intel and AMD devices) or is specifically describing Apple Silicon.

I think the situation with Apple Silicon is that the GPU L2 lines have state, which tracks which lines have been modified, and the SLC cache has knowledge of line addresses in the L2. This means that, at the very least, it's possible at a synchronization event, not to flush the whole of L2. Rather we can

- write back the L2 lines that have been modified, and
- the SLC can inform the L2 of which L2 (read-only) lines were modified by another agent (eg the CPU) and the GPU L2 can then invalidate those lines, while
- leaving valid all the lines in the GPU L2 that were not touched.

That's already a nice improvement over a full flush, and if we only perform these tasks at synchronization points, rather than moving lines around on every L2 modification, we may save some power.

We can do even better if the API allows us to mark certain resources [ie blocks of memory, ie address ranges] as "temporary" or "GPU-Private" or similar type flags. If we know that these resources will never be accessed by something outside the GPU, then we know that even writes to these specific lines do not need to be written back to SLC. They can persist in the GPU as long as makes sense, and then just be discarded. Apple also does some of this. If you think about the details, what you want is something like

- a way to mark a line when first referenced as being a "GPU-only" line, either when the data is read from DRAM or if some data is written by the GPU.

- if such a line has to be evicted from L2 to SLC (to make space for other data) ideally the "GPU-only" flag travels with it to SLC, and back when the line is re-referenced by the L2.

- at the point of forced L2 coherence, lines with this flag within the L2 can simply be ignored, no need to send anything about them to SLC. Depending on other details (maybe a second flag) we can either invalidate the line at this point or keep it around for reuse (eg maybe it's a texture we synthesized that we keep using from frame to frame).

It's interesting to compare this to the competition.

As I understand CUDA, "traditional" CUDA has only a heavyweight barrier between kernels; between kernels everything is flushed to HBM (the equivalent of Apple's barrier between Command Buffers) and there is nothing lighter-weight, no equivalent of kicks that synchronize cores within the GPU, but don't synchronize with any sort of RAM or the CPU.

With Hopper, CUDA introduces the ability to launch a sequence of kernels that are "tied together". The only examples I have seen of this are based on a kernel launching a subsequent dependent kernel, but I don't know if that's the only way you can force this tied-together ability. This sequence of kernels is essentially like an Apple sequence of Tile Shaders, a sequence of kernels that execute different code, but share the same block of data at the equivalent of L1 cache level. As you would expect from the target markets, the Apple scheme is (for now) optimized for graphics, with only limited compute

capability (basically the ability to execute a Compute Shader partway through a sequence of Tile Shaders, while the nVidia scheme seems (for now) optimized for general compute and AI, with the graphics tools not yet exploiting it).

OK, just to be absolutely sure you have the idea, let's go through this one more time, So suppose the GPU requests data that it has never loaded before from address A. The load will miss in GPU L1 and GPU L2 and will go out to the SLC. The SLC is coherent with the CPUs, so the most recent version of the data, even if it is in some CPU cache, will route to the GPU. Great.

But now suppose

(a) the CPU modifies data at address A. The GPU again requests data from address A. This (whether it hits in L1 or L2) will not be that updated CPU version.

The programming contract is that during the period that the GPU “owns an address range” the CPU is not supposed to mess with it. Practically this is not too hard, you just use a model where the CPU does its part, fills up a buffer or two, hands them over to the GPU, and doesn't touch them again until

(b) the GPU writes out new data (either overwriting the buffer at address A, or creating a new buffer at address B). In either case, as far as the hardware is concerned, if the CPU tries to access data at addresses A or B it will see stale data.

The essential contract is that

The GPU begins a block of code (there are many different words Apple and GPU documentation use for different sized blocks of code, and we don't want to get lost in details, so let's just this block of code a “job”) with the assumption that none of the relevant data is in the GPU caches, and so all loads will route through the SLC and will get the latest version of the data.

Then, for the duration of GPU execution, this data is isolated from the CPU. (In principle some data could, for example, overflow to the SLC or even DRAM, but regardless the CPU should not be trying to read or write it.)

Finally at the end of the job the GPU flushes all its caches, all modified lines transfer to the SLC, and we're back in sync with the CPU.

Let's call this sort of operation a “major flush”. As we have seen, it corresponds to the boundary between Command Buffers.

Now think about the smaller segments of a “job”. Suppose (in a hand-waving very simple way) we describe the process of creating a 3D image as two parts, and our simplified GPU has only two cores:

- step A is we run through an array of geometry (points in 3D space forming triangles which in turn form shapes) transforming each 3D co-ordinate to a 2D location on the screen (which will involve rotation, scaling and suchlike)
- step B is for each on-screen triangle we perform texture lookup to fill in the triangle with its color/pattern/texture and write it to window buffer

We perform the first step by splitting the array in half so each of the two GPU cores does the rotation/scaling/projection arithmetic on half the geometry.

We perform the second step by splitting the screen in half so each of the two GPU cores handles the triangles that are in its half of the screen and performs texture lookup.

The important point in this is that the triangles handled by the first step may not be processed by the same core for the second step; the criteria for splitting the work differs for the two passes. This means that we have a similar sort of issue where we do not want stale data in each of the L1's (which are per GPU core and not coherent).

But note that the L2 *is* shared by the GPU cores, and so at the end of the geometry stage we can perform a “minor” flush which flushes L1's to L2, but does not need to flush L2 to SLC.

Note that flushing means both the obvious writing back modified data and the less obvious marking as invalid lines that were only read (because eg the buffer from which they were read could now be changed by the CPU before that buffer is to be read again).

As we have seen, this corresponds to the boundary between dependent kicks.

So at the software level, we have a situation where the instruction stream will, depending on how a block of generated data is going to be used (by the GPU again, or by the CPU) include some kind of flushing command of a block of work. The details of those commands and who is responsible for them (the Metal Shading Language? the GPU driver? the GPU firmware? the developer?) are sometimes important questions for developers but not for us. What matters for us is that, occasionally, flushing of either all GPU caches, or just L1 caches to L2, are required.

Given all this, can things be arranged so as to minimize the costs of these “flush-like” operations?

For example, can we somehow tag lines as being associated with Job 1 rather than Job2, so that rather than flushing all lines, we only flush the lines associated with Job 1, leaving untouched the lines of the irrelevant Job 2 (which maybe hasn't completed yet, or maybe doesn't require flushing at all)?

Likewise we could imagine richer flags for the flush commands so that, at the end of a job, we flush out modified lines from the cache but don't have to flush out read-only lines (because we have guarantees that no-one, neither the CPU nor other GPU cores, modified those lines over the previous job)?

Alternatively, or in addition, can we somehow spread out flushing the appropriate lines, in the background, so that we can start the next job sooner? Rather than waiting till the end of a job and flushing the entire L1D, perhaps we can detect we are close to the end of the job, and start writing back lines to L2 at a rate that will leave us with a close to empty set of lines by the time we finish the job? We'll see a set of patents associated with this issue of “partial coherence” between the CPU, the GPU L2, and the multiple GPU L1's.

I recommend at this you lie in bed and think through the various issues I have just described. This stuff is not obvious, especially if you come from a CPU world where everything is coherent all the time, so you need to think of, first how the worst case (full flushing) has to work, and then how various specialized cases could be made more efficient.

Unfortunately the Apple patents (or anything produced by anyone else like AMD or nVidia) are never very clear about exactly what is promised in terms of synchronization at this high “cross-kernel” level.

Certainly what the Apple patents say sometimes looks to me suboptimal, so perhaps this has changed over the years?

We considered things at a high granularity (what's promised about synchronization between the CPU and GPU in terms of kernels starting and ending). This is, conceptually, memory ordering, but at an "API"-level.

Now consider things at low granularity, in the same way that we think about CPU memory ordering.

The obvious cases are if we have code that could be executing at unpredictable times relative to each other, and unpredictable locations.

Recall that a threadblock (up to 32 warps) will all execute on the same core, but the individual warps (each 32 lanes wide) can execute in any order relative to each other. So if we require an ordering (ie no lane proceeds until the entire threadgroup has reached this point; or this warp needs to read data written by some other warp in the threadgroup) we will need some sort of barrier.

This is kinda-sorta like an even weaker version of a CPU memory model. The standard CPU weak memory model (eg ARM) is essentially that if CPU A writes to addresses X and Y, there is no guarantee (in the absence of a barrier) that these will appear visible in memory in the order X changes first, then Y changes.

The GPU model kinda extends this to a much "wider" version of simultaneity and ordering. The threadblock *conceptually* executes simultaneously across every lane on an imaginary 32×32 (32 warps of 32 lanes) wide core; but there is no promise (in the absence of a barrier) that if any of these 32×32 lanes reads something written by an earlier instruction it will see that changed value (in part because there is no obvious meaning to "executed earlier", even for warps from the same threadblock).

Even for a *single* warp there is no guarantee that every load, or every store, of 32 lanes of the warp, execute in the same cycle.

- The 32 logical lanes of the warp may execute on, say, 8 physical lanes of load/store, over four cycles.
- Some of the lanes' load/stores may hit in cache while others have to go out to DRAM, so they may complete at wildly different times.

This should be obvious when you understand how the threadblock is executed, as different warps on different quadrants, all occasionally pausing to give time to a different warp.

So we have a weaker memory model (in some kind of sense we're not promising ordering even within a single core, to the extent that we imagine these 32×32 lanes forming a single virtual core.)

This is obvious. But now consider more GPU-specific cases:

First remember that we have a core of four quadrants, each executing 32 lanes. Suppose that the first and last lane of a warp both store (different) values to address X. Who wins?

Or suppose that the first lane of two warps executing on two different quadrants of the same core both store (different) values to address X. Who wins?

In both cases your response should be that for this to happen is a race condition and a programming error! The hardware should not make any attempt to bother trying to do something "sensible", it

should just demand that you write a better program, eg by using atomic memory instructions, or ensuring that every lane always writes to a unique address.

In the past even weirder things were possible, when GPUs used very different read and write paths to different caches. But this is history; it no longer seems relevant to modern designs, though you may want to bear it in mind if you try to understand GPU designs or even some aspects of APIs from before ~2010.

At this point you might enjoy watching (2019) <https://www.youtube.com/watch?v=VogqOscJYvk> *The One-Decade Task: Putting std::atomic in CUDA. - Olivier Giroux - CppCon 2019*, which is a discussion of some aspects of these issues.

(BTW Olivier Giroux is no longer at nVidia but has been Director of GPU Architecture at Apple since early 2022...)

Kernel Flow

So we have the basic idea that a GPU is given a large number of kernels which are executed simultaneously across multiple lanes (32) of quadrants (four per core) of multiple cores (4 for A14, 8 for an M1, 16 or 19 for an M2 Pro, and so on). But there are many details to this.

Once again I'll somewhat repeat the overall story.

Starting at the highest level we have multiple *independent* instruction streams. Independence means that these streams have no to very little relative synchronization so can be handled almost completely independently. Each such stream may, for example, come from a different program. For the most part dependence means memory dependence, ie the first kernel writes out data to global address space which is then read by the second kernel. You could, in principle perhaps, imagine something where we want the second kernel to execute after the first kernel for some reason even though they have no shared data, but that's so unusual as not to be worth bothering about.

We've already pointed out that a single kernel consists of multiple threadblocks (up to 32 SIMDs each 32 lanes wide, so a threadblock can correspond to 1024 lanes of instructions).

So the simplest model for operating the GPU could consist of executing one kernel at a time, until that kernel is done, then starting the next one.

Within that model, simplest is to launch one threadblock onto a core at a time, and execute that until it is done.

On the plus side, this is simple and probably makes close to optimal reuse of memory traffic. But

- clearly it leaves a lot of dead time while we are waiting for things to complete;
- it can't do much to overlap the execution of different parts of the hardware (so that the vertex processor and texture unit are both working hard at the same time)
- even within the code of a single threadblock, it tends to run the GPU core as "phases" (ie all the lanes,

of all the active threads, all want to do the same thing at the same time, like they all want to access memory)

We can do a little better if we schedule a few threadblocks (still from the same kernel) on a core, allow the first to get going, then mix in threads from the second then the third threadblock. Now, hopefully, we get a more even mix of instructions, some to memory, some compute, some texture, etc. Once we have this machinery, we are probably also now flexible enough that whenever a threadblock ends we can start up a new one right away, so as the threadblock draws to a conclusion and there are only three or four SIMDs still executing, we have already started a few other threadblocks, they are partway through, and all the potentially active threads on our core are fully filled.

Next up is to allow for the execution of different kernels simultaneously. The weakest form of this requires the kernels still to come from the same single process (which, among other things, means we can maybe ignore an MMU and security issues). But at least now we can overlap threadblocks hitting the vertex hardware with threadblocks hitting the texture hardware. And once again, now we can avoid dead time as we wait for the last few straggling SIMDs at the end of a kernel by overlapping the execution of the next kernel or two.

Of course even better is to allow simultaneously different kernels from different processes; but now we do have to take MMU issues seriously.

Even in this case, however, we may perhaps not allow for pre-emption. In other words, once we fire a threadblock, we may have to wait until it is finished before we can cleanly start up a replacement. (There will be mechanisms to handle catastrophically out-of-control threadblocks, like a reboot of a core; but clearly that's not a great solution...)

At this stage of evolution the compiler may insert “pre-emption” points in the code where the GPU can be pre-empted, but whether this is feasible or not depends on exactly what the code is doing; this scheme may not help pre-empt a tight inner loop.

So the final stage is full pre-emption on any instruction, like a CPU.

One of the more important aspects of this evolution is the question of “what are you optimizing for?”.

- Optimizing for a single *kernel* at a time is what many (most?) academic papers do. You can learn a lot from this; but few companies want to do exactly this.
- Optimizing for a single *process* (aka “game”) does present a desirable target for many companies, though probably not for Apple.
- Optimizing for multiple *processes* is, of course, hardest, because you continually have to balance the obvious win of memory locality (finish a single kernel as much as possible before starting the next one) against the obvious win of execution unit diversity (execute computation while another kernel is calculating textures). One thing you’ll notice as a constant Apple pattern is how strongly they seem to prioritize execution unit diversity, and the scheduling for this.

Within a stream, and if you are implementing traditional graphics, the baseline Metal API provides a synchronization model that tries to strike a balance between performance and ease of use.

Some parts of the later pipeline will rely on data created earlier in the pipeline (for example

fragment shaders cannot run until geometry has been processed; or using an image as a texture needs to wait until the texture has been created). Metal provides a way within MSL (the Metal Shading Language) to mark the data resources that are created by different kernels, and will provide automatic synchronization for these data resources. This may be implicit (you mark the data output by a vertex shader, and Metal knows that fragment shaders depend on this output) or slightly more explicit (you mark an output image buffer in a particular way, then mark the later use of a texture as being that same image buffer).

This is essentially a fairly easy declarative way to describe your graph of data dependencies without having to ever think about parallelization issues. It's specifying which kernels are dependent on earlier kernels (ie previous kernel writes data to be read by later kernel) so that the GPU automatically enforces

- second kernel doesn't execute until first kernel has finished AND
- all the data created by first kernel has been flushed out to L2.

The corollary to this sort of ease of use is that forcing kernel dependencies hurts performance if those dependencies are not required. If your code is generating implicit kernel dependencies by Metal trying to be helpful, you might want to look at ways to restructure your code so avoid that.

the command stream structure

The result of this work is the creation of a command stream that consists of three elements:

- a header
- pre- commands
- kick
- post- commands

The pre- and post- commands are the automatic synchronization that Metal provides to meet the API as earlier described. They may consist of waits to ensure that certain kernels do not begin until earlier kernels are complete, along with data transfer to ensure that data created by an earlier stage and present, for example, only in L1 or in Local Address Space, is moved as appropriate (from Local Address Space to L1, or from L1 to L2) so that subsequent kernels can see it.

The kick mechanism operates at a very high level, in firmware, ensuring that entire groups of kernels are delayed relative to each other.

On the one hand, this means such dependent kernels waste no GPU time waiting for each other, the wait happens at the Firmware level outside the GPU proper. Likewise the data transfer is implemented by “data movers” rather than the GPU pipelines.

On the other hand, this wait is very coarse grained; so you want to bypass if your code is such that you can usefully get some part of a kernel to do some work even if another part of the kernel is required to wait for earlier code.

As the system has evolved, one thing you will notice is how a variety of developer-level synchronization mechanisms have been added that can be used within a kick, so that one task within a kick can make use of a lightweight synchronization that (on the GPU) will delay execution of one task relative to

another. These mechanisms demand more effort from a developer, but allow the developer much more control, either to do things outside the baseline Metal graphics model, or to overlap work from multiple kernels in a more sophisticated fashion.

Likewise the system has evolved so that these parts that Apple controls (queuing up kicks and pre-/post- commands) are more efficient, scale to larger pools of work, and can do more in terms of overlapping some elements of work (like moving in the next kernel's instructions simultaneously with clearing out the caches from the previous kernel)

outermost scheduler

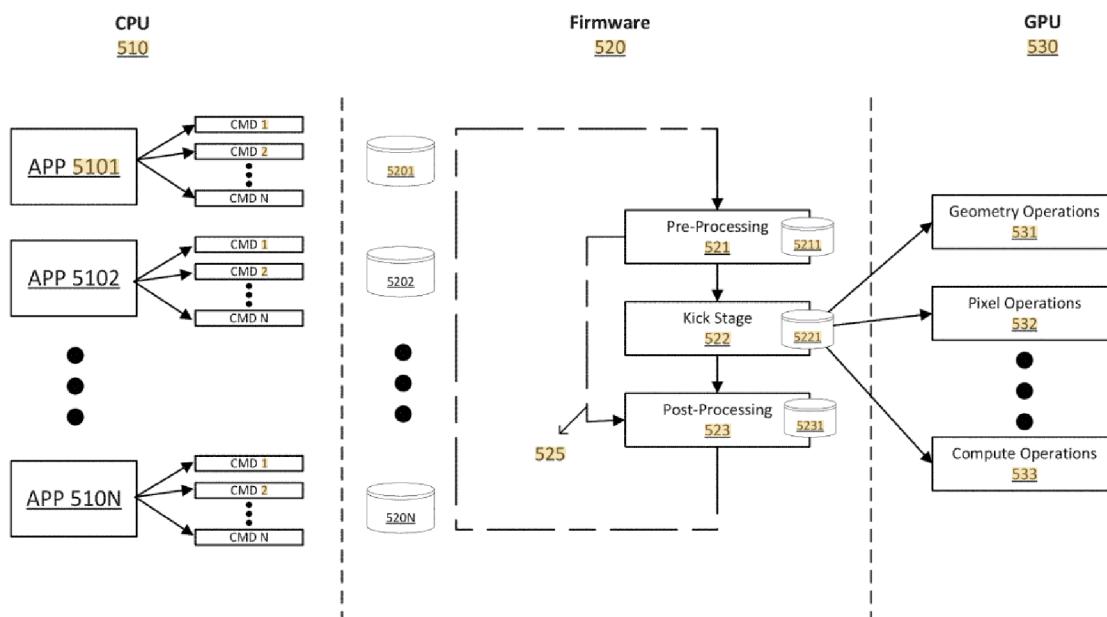
At this point we have enough background to start understanding some patent details.

(2018) <https://patents.google.com/patent/US10719970B2> *Low latency firmware command selection using a directed acyclic graph* discusses the instruction flow (ie scheduling) at the highest level.

To recap, multiple apps are each generating multiple streams of “commands” for the GPU. These commands correspond to large blocks of work (kicks or kernels) together with various types of dependencies between some, but not all, of the commands.

The commands flow through the GPU driver, which is fairly lightweight, to be processed by “Firmware” and the GPU. Firmware refers to

code running on the GPU’s companion processor, a small Apple-designed ARM64 processor running a small custom Apple OS.



“Kick stage” corresponds to “real work” that is sent to the GPU. Pre- and Post- processing implement the dependencies, either waiting for an earlier kernel to complete, or executing some sort of cache data movement to ensure that a later kernel sees earlier created data, and perform some degree of resource allocation/deallocation. The primary resource allocation, I think, is some virtual address

space to provide the kernel with stack space; maybe also the behind-the-scenes graphics tasks (ie code written by Apple, not by developers) allocates virtual address space at this stage? But this is too early to allocate the obvious resources that are per-threadblock (Scratchpad) or per-warp (Registers).

So what's fed to the Firmware is high level instructions along with dependencies between these high level instructions. (Remember high level means something like, conceptually "function that will calculate an image labelled X" followed by "pixel shader that will read image labelled X as a texture".) Firmware then does various things with this stream. One is to implement the work of pre- and post-processing (enforce dependencies; move data between caches), the second is to break up the high level instructions into lower level instructions that are fed to the actual GPU. (This does not have to be especially dramatic or expensive; it's more like the Firmware gets given a stream of packets. At the head of the packet is the data the Firmware requires, like the dependencies and their types, and a payload length; then what's sent on to the GPU is the payload stripped of this header.)

As we have seen, at the highest level, the GPU companion processor has available a pool of tasks that have to be performed, both the developer supplied tasks and companion tasks like flushing caches; and a set of dependencies between these tasks. The goal is to ensure that whenever resources become available on the GPU, a task is available to execute subject to all its dependencies having been satisfied.

This goal is performed in two stages:

Firstly, based on the dependencies, building a graph of tasks, each of which is labelled by how many earlier tasks it depends upon. From this graph it's easy to extract the set of tasks that have no remaining dependencies. These are placed in a separate set of queues from which they can rapidly be dequeued based on priority (and perhaps other criteria, like whether they are expected primarily to move memory around vs using Textures vs performing computation) so that, when GPU resources become available, we can easily find a close to optimal task that's immediately ready to execute.

The patent itself deals with the question of how do we enforce the dependencies between Kicks? This is essentially about building in software, from the stream of Kicks, a dependency graph (which changes every time a new Kick is added or completed) and some tables which allow for rapid lookup, once a given Kick completes, of which are the dependent Kicks which now have all their dependencies fulfilled, so that they can now be treated as executable as soon as the opportunity arises.

Now the above description slid over the issue of how the dependencies between some kernels and not others are established and communicated to the Firmware. This is described in (2018) <https://patents.google.com/patent/US20200104968A1> *Graphics Driver Virtual Channels for Out-of-Order Command Scheduling for a Graphics Processor*. There will be some kernels submitted to the Driver (and processed at this stage on the CPU) which are naturally independent, for example those associated with different processes.

But of those kernels that may perhaps have an implicit Metal dependency (eg through labelled resources used by multiple kernels), the Driver will establish dependency.

The Driver then creates separate "Virtual Channels" with the Firmware, and a sequence of dependent kernels, in essentially dependency order, is sent down each channel, with the understanding that

kernels in separate Virtual Channels cannot have any dependencies on each other. This presumably reduces the amount of work the Firmware needs to perform as it builds its dependency graph (based not on these highest level commands but on the micro-command including pre- and post-commands), since it only has to worry about the interactions between the much smaller set of micro-commands associated with any particular virtual channel.

This is the way the patent is described, but it cannot be the entire story. Think about the boundary between Command Buffers, which require some sort of entire flush of the GPU L2 to synchronize with CPU memory. This is a “dependency” across all kernels, so it can’t be communicated by a scheme that’s based on separating independent kernels.

There must be an additional (undescribed) mechanism for indicating the Command Buffer boundaries.

Another unexplained issue is how do we handle long running (longer than a frame) compute tasks that are executing alongside graphics work. Are they preempted every frame, put on hold while the L1D’s and L2 are flushed, then resumed? That seems sub-optimal! Or is it possible to segregate such work from graphics work, eg by having some compute buffers opt out of the Command Buffer boundary implicit total synchronization?

implementation of dependency tracking

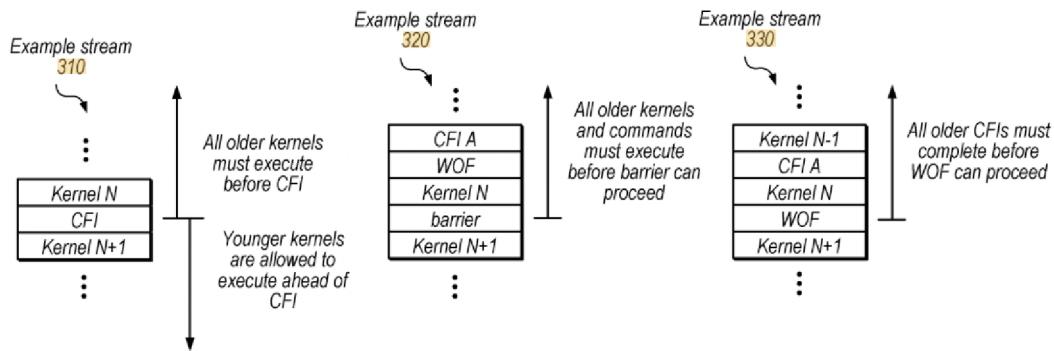
So far we have established that ultimately the driver submits a sequence of kernels, that dependencies between kernels are established, and that streams of independent kernels are transferred to the GPU.

But what about the kernels that are within a dependent stream? The command stream essentially looks like kernels (blocks of code associated with a grid of a certain size, that will be executed on a multi-lane basis) separated by “control commands” that perform housekeeping for the kernels. Examples of these include

- barrier (wait until every thread in the previous kernel is finished before you start the next kernel)
- cache flush and invalidate (CFI)
- wait for cache flush complete (WOF – Wait on Flush)

The semantics of these mean that there are situations where the previous kernel needs to flush a cache, but the next kernel does not care about when that finishes (ie it does not require a “wait for cache flush complete” instruction). For example maybe it does not make use of this particular cache or address space? So we want to take advantage of this (when possible) by starting the next kernel without waiting.

But it’s not a completely trivial process because we may have a situation where kernel K0 wants to flush the cache, kernel K1 does not care, and kernel K2 needs to wait until that cache is flushed. So there’s a kinda propagation of dependencies that needs to be tracked across multiple kernels. Overall we see another pattern we’ve seen repeatedly in Apple designs; define precise barrier-like commands rather than vague *stop the world* “synchronize” type commands, define exactly what each barrier-like operation demands, and allow work that is not enforced by the barrier flexibility is moving relative to the barrier.



(2018) <https://patents.google.com/patent/US10475152B1> *Dependency handling for set-aside of compute control stream commands* describes a way to take advantage of this without requiring too much overhead. One naive way to do this would be something like, every time either a control command or a kernel completes, we iterate through the set of commands and kernels seeing what we can or cannot do next. But if we are willing to spend a few extra bits, we can create some auxiliary structures pointing between various tables so that once a command or kernel completes, we only have to follow the obvious pointers to establish what commands/kernels we can execute next. If you're interested, the patent describes exactly how these tables and bit-“pointers” work.

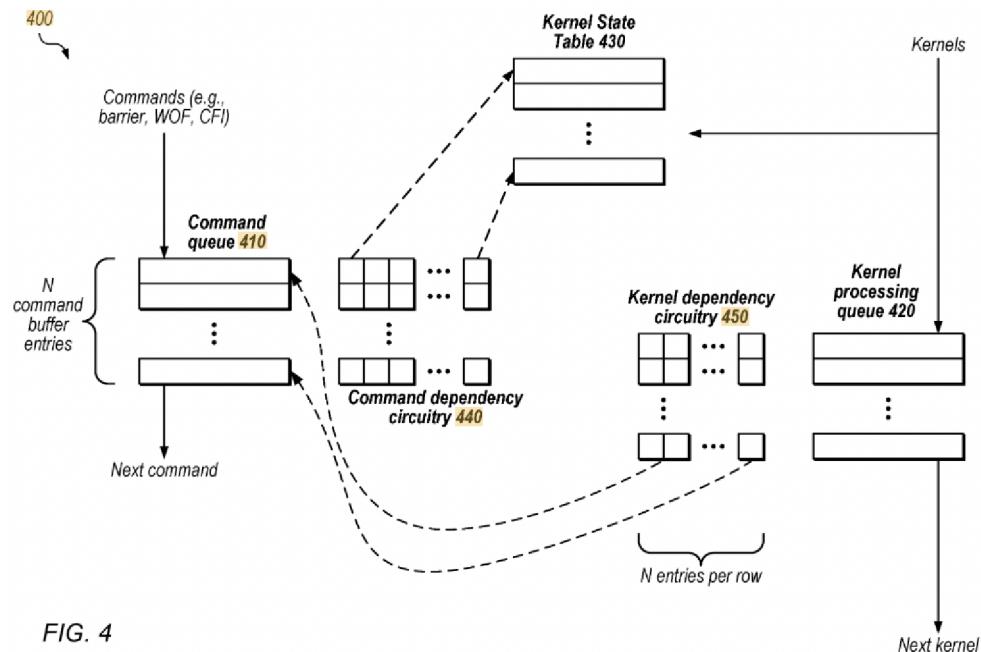


FIG. 4

scheduling at a slightly lower level

The command stream as described is executed outside the GPU proper. From this stream, if we strip out the pre- and post- commands, we have a stream of kicks, which can be thought of as a stream of kernels to be executed.

A kernel can correspond to a very large grid, but kicks may also have dependencies at a fairly fine-grained level.

Specifically Apple encourages graphics code to be written in such a way that a kernel (which ultimately covers the entire screen) corresponds to subgrids each of which are a tile (think something like a 32×16 pixel rectangle of the screen), and these kernels feed into subsequent kernels. Ideally one kernel may take in the results of a tile, manipulate it, and generate new results for that tile.

When this is feasible, in some sense you want

- the grid of that kernel to be split into tile-sized elements
- which are interleaved with tile-sized elements of the next kernel
- in such a way that each kernel is directed to the same core as its predecessor tile

This doesn't exactly map into the way we have describing Compute kernels and the implied memory synchronization. Or more precisely, there are ways to use the machinery I have described so that some dependencies you may have worried about may not be an issue.

Suppose for example that we indicate that

- task A will read from memory via a read-only buffer that is not used by any other kernels in this command buffer
- task A will write only to its tile (ie to its Scratchpad storage)
- to be followed by task B which will read from that tile then write back to it, but doesn't read from global memory

Under these circumstances, we no longer need to have any sort of cache synchronization passes before A, between A and B, or after B.

The point is that splitting a kernel into smaller executable elements (eg workgroups, as blocks of up to 1024 "lanes" of work) is not a trivial process. Different types of grids have different requirements [large compute grids to be spread over multiple cores, versus fine per-Tile grids where we try to sequence successive work to the same tile on the same core], there may be dependencies (of different types, perhaps only at the tile/Scratchpad storage level, perhaps also via global memory with L1D cache writeback implications) between successive kernels, and of course you're always trying to ensure load-balancing (no core is going unloaded) along with low latency (no core starts executing and then can't be reused for many minutes while it grinds away at its work!)

So the stream of kicks is broken up into a stream of smaller "jobs", each job marked with various details (perhaps it needs to be "tile-interleaved" with the next job, perhaps it can be trivially spread across every core with no restrictions, whatever).

The initial processing of this stream probably also happened in firmware, at a single location, where the grid of each kernel was broken up into large batches each then broken into workgroups, with the workgroups then enqueued on individual cores for later execution.

This procedure of breaking large grids into batches of smaller work has changed fairly substantially from the earlier Apple cores till now, to incorporate the existence of the M1 Ultra and the fact that work needs to be scheduled optimally across two separate chips.

One question to bear in mind as we go forward is whether the GPU is ever throttled by the performance of its (not especially powerful) companion core. As the design has evolved, we've seen more of this lower level work distributed across the entire GPU. The details are never specified, but essentially the idea now is to break the large job into "batches" that are dispatched to a dispersed set of elements (the size is unclear, but let's imagine say 4..6 cores). These local elements then break up each batch into workgroups, enqueue the workgroups on individual cores for later execution, and track the completion of the entire batch.

At this point we have arrived at a given core having a queue of workgroups (the kernel of a grid, along with some rectangle of work to perform within the grid, of size up to 1024 elements). For example the common graphics situation would be that the grid is the size of the screen/window, and a tile size of 32×16 corresponds to 512 pixels of that screen/window.

These workgroups are queued up within the core. However most of the workgroups are note yet "active".

At any given time the workgroup will be executed one SIMD/warp (32 lanes wide) at a time, and

within each quadrant of the core anywhere from 6 to 24 warps, from some number of workgroups will be active. The number of simultaneously active threadgroups depends primarily on how much Tile/Scratchpad storage each threadgroup requires; the number of simultaneously active warps depends primarily on how many registers each warp requires.

Within these active warps the hope is that whenever a particular warp stalls (eg loads from L2 or even DRAM) the other active warps can continue execution.

When all warps of a workgroup complete, then

- one of the inactive workgroups can now be started, AND
- the higher scheduling machinery (the local batch processors, then the global batch processor, then the firmware) needs to be informed. These higher level entities need to know when their batch of workgroups, or the entire kernel, is complete. Based on *that*, we can then track when dependent commands (eg flushing out the L1 cache to L2), and then dependent successor kernels, can begin.

clauses

The kernel instruction stream would seem to be as low as we can go, but in fact that stream consists of instructions grouped into “clauses”. Clauses seem to be an idea from AMD/ATI.

The rough idea is that “execution” clauses are separated by “control flow” instructions. Control flow instructions modify the single PC that is shared across all lanes of a warp; execution clauses execute the bulk code that applies to all (possibly predicated) lanes.

There are a few different types of clauses that each can hold only specific instructions, for example ALU clauses (the most obvious), vertex fetch, or texture fetch, perhaps memory load/store.

Clauses reduce energy by allowing for

- simpler sequencing of most instructions. They are something like traces (a short straightline run of instructions) that can be executed with minimal control overhead (no need to track control flow or even the exact PC, you just start at the beginning of the clause).
- aggressive reuse of decoded instructions without having to move them around the chip. Instructions flow from L2 (shared by the entire GPU) to a per core L1 to multiple small *clause caches*, each associated with a different unit in the core (texture, vertex, compute, etc) out of which they are executed. Once in a clause cache, the decoded instructions of the clause can be used repeated, either within loops or by repeated warps of the same kernel.

indirection

Finally, at this level of setting up background concepts, let’s discuss the meaning of *indirection* for GPUs.

There are two distinct concepts here that sound similar – don’t get confused! For example consider these two patents:

(2017) <https://patents.google.com/patent/US10657619B2> *Task execution on a graphics processor using indirect argument buffers*

(2018) <https://patents.google.com/patent/US10269167B1> *Indirect command buffers for graphics processing*

indirect argument buffers

Very simply, an indirect argument buffer packs multiple data items (eg some textures, some material properties, etc) in a single large array. This array can then be passed around into various GPU calls, rather than passing separate arguments.

The shaders know how to walk the data structure to find what they want.

This conceptually is familiar and obvious to CPU programmers; what makes it look a little bit messier and unfamiliar on the GPU side is that, for optimal performance, you need to indicate in some way the labels and usages of each item packed into the buffer so that the GPU can perform all the resource/kernel dependency and cache management we have already described.

indirect command buffers

Meanwhile an indirect command buffer allows you to construct a sequence of GPU commands on the GPU, rather than on the CPU.

Now those are basic definitions. What do they mean? Here's my super-simple example for how I think about them.

Consider a complicated 3D scene. Ultimately this is represented by a scene graph (multiple sub-graphs representing individual items, with their shape, texture or material, etc) all aggregated into a larger graph indicating the locations and orientation of each item. You may, for example, have one "warrior" object, replicated to appear fifty times, in fifty different locations and orientations, across the scene.

So

- in the simplest version of a graphics API, **on the CPU** I walk this scene graph and for each item of interest I make a draw call into the graphics API giving a location, a texture, whatever. There's a lot of state attached to each draw call that has to be moved between the CPU through the graphics system to the GPU.

- next up in sophistication, I can essentially flatten that scene graph to a single long array and pass that array as an argument to every draw call. I am still walking the scene graph on the CPU (and now I am also doing some walking of that same scene graph on the GPU) but I am passing much less state repeatedly from CPU to GPU.

This is what I get with indirect argument buffers.

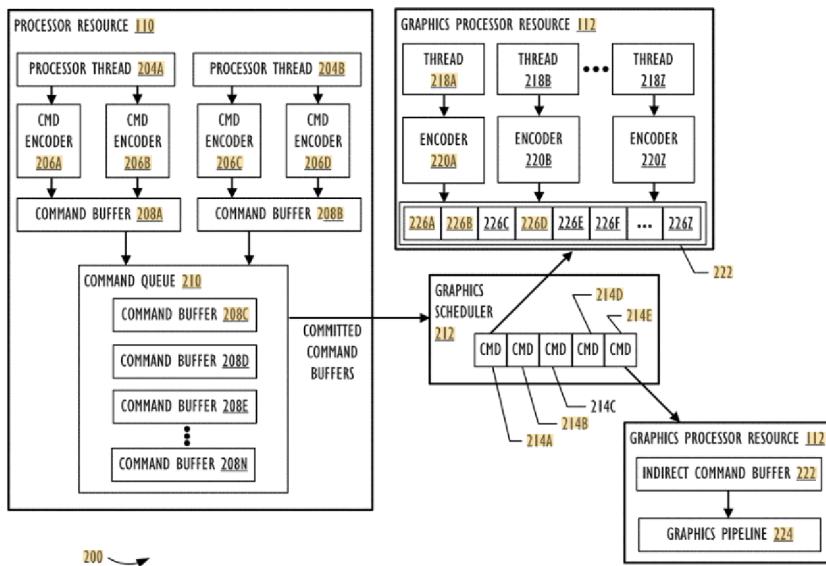
- best of all, suppose that I walk the scene graph **on the GPU**. This is implemented, essentially, by providing the GPU with an indirect command buffer, and executing a "produce" command which walks the scene graph and fills the indirect command buffer with appropriate commands. Once the indirect command buffer is complete, a "consume" command is executed which starts executing the commands in that indirect command buffer.

Now all the scene graph walking and command construction work is done on the GPU without CPU involvement, and we also don't have to pay the costs of moving the command stream and associated data onto the GPU (even with unified memory, this still requires moving between CPU and GPU caches

which costs energy).

This is essentially the same as the Dynamic Parallelism feature that nVidia added to Kepler in 2012.

These two diagrams give something of the idea:



This is more or less the standard model. CPUs encode commands through one or more command encoders into command buffers. Those buffers are committed to a queue (meaning that the buffer is considered complete, it can no longer be modified). The command buffers flow from the queue through the GPU firmware (here called Graphics Scheduler) which breaks them up into pieces to be sent to GPU cores.

But the advanced model allows one of those commands, say command 214A, to be a produce command, which tells each lane of the GPU to execute a kernel which walks the scene graph in some co-ordinated way and encodes draw instructions based on whatever node that lane finds. Those draw commands are placed in buffer 222. A later command, like 214E, can switch execution to consume the commands placed in buffer 222. Naturally (among other things) this is going to require some new form of synchronization for the highest level scheduler we earlier described, so that the execution of 214E is delayed until 214A indicates that it is completed.

(And then some sort of synchronization is performed to flush buffer 222, most of which will be sitting in an L1D cache, out to GPU L2. At which point, what?

Does the firmware core share L2 with the GPU? Does it only communicate with the GPU via SLC? Has this changed with time?

It might seem like a good idea, overall, to have the GPU controller share an L2 with the GPU. But the GPU controller is on a separate power island from the GPU (so that it can stay awake and in communication with the rest of the system while the GPU sleeps). Maybe put the L2 on that same power island? Not obvious what to do, is it!

Everything is more complicated than you think, when you start to consider the details! And you start to realize why something that “seems obvious” and “should have been there on day one” actually requires considering a whole lot of issues you never thought about, like what sort of communication channels exist between the GPU proper and the companion core controlling it...)

Regardless, somehow some combination of flushing moves buffer 222 from one or more GPU L1D’s to a location where it is visible to the Firmware, which can then start executing from said buffer 222, treating just like say buffer 208C generated by the CPU.

This is indicated by FIG 3 below. One can even do fancier things like repeatedly execute that set of commands (as in FIG 4).

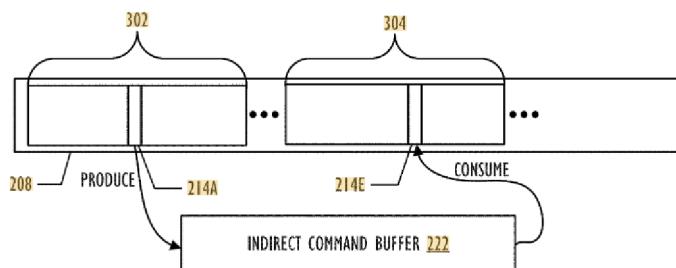


FIG. 3

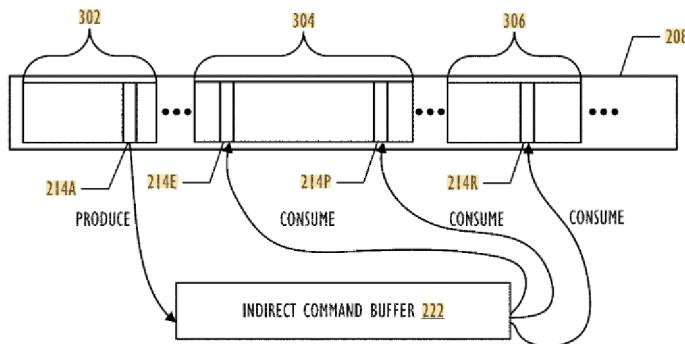


FIG. 4

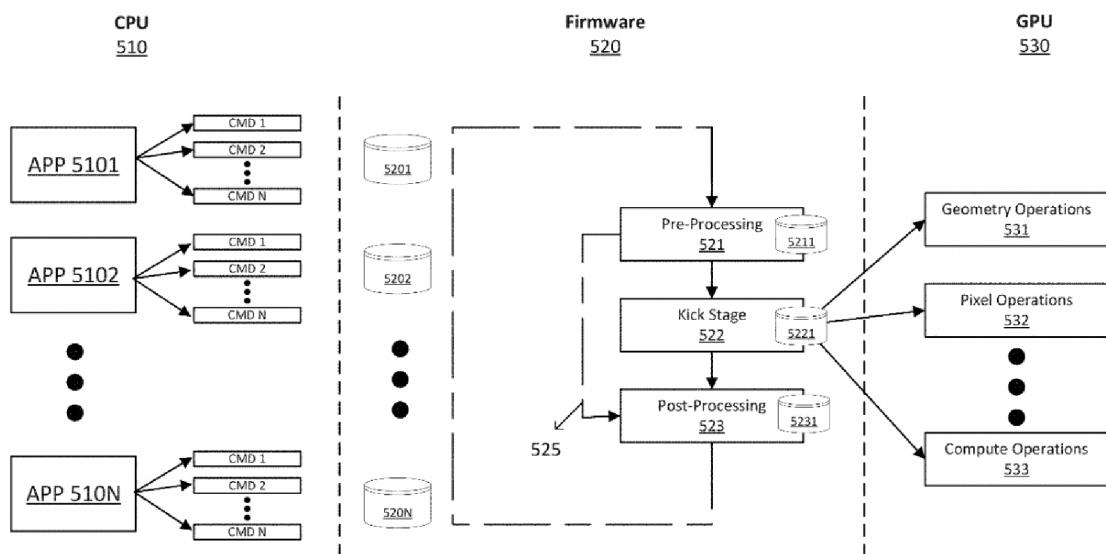
You can read something about how Indirect Command Buffers are used at
<https://nilotic.github.io/2018/09/03/Metal-for-Game-Developers.html>

Kernel Flow revisited (move all dependency parsing to firmware) - 2019

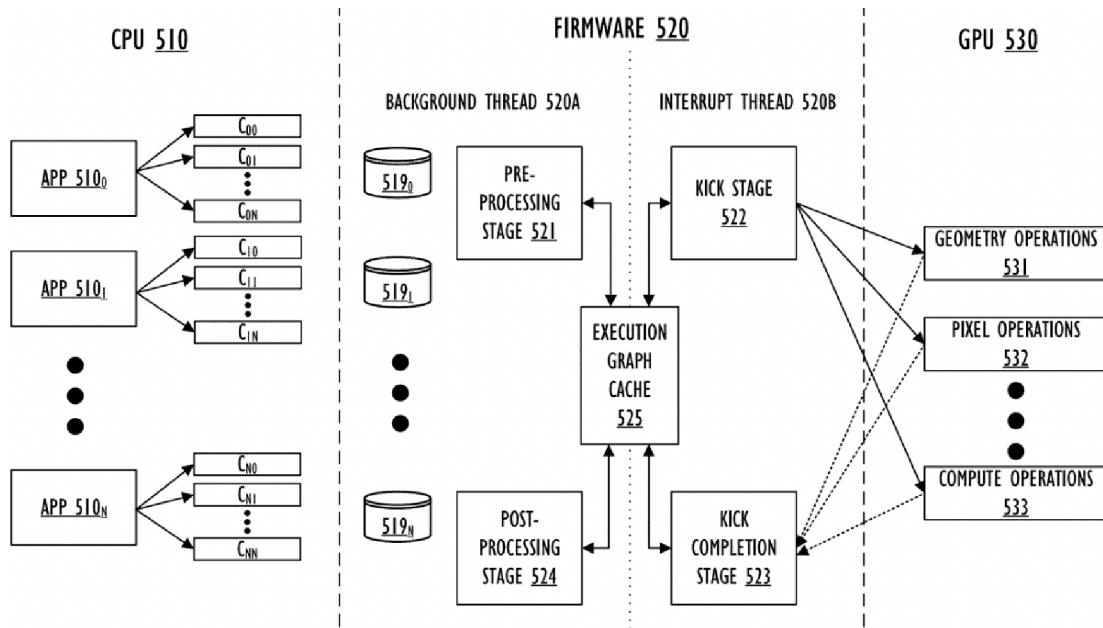
The 2018 patent describing scheduling by the Firmware (creating a datastructure of dependencies so that it could easily track when any particular kernel no longer had any dependencies and could be scheduled) is updated in (2019) <https://patents.google.com/patent/US11436055B2> *Execution graph acceleration.*

Indirect command buffers mean that some work is now created on the GPU and fed back to the firmware, without passing through the driver on the CPU. So some of the work involved in establishing dependencies between kernels now has to happen in the Firmware...

The overall idea remains the same, but many details have changed. Compare the previous diagram



with the new diagram



The firmware now operates two threads. A low priority thread

- accepts command buffers,
- parses them,
- generates a dependency graph,
- executes “overhead” commands between kernels, like flushing caches.

A high priority thread services interrupts that communicate that a previous task has completed. In response to this, the interrupt service routine decreases the dependency counts of all the affected kernels, and starts the execution of any new kernels that may have become runnable as a result. Later work generated by this completion (the post-processing stage of things like cache flushing) will be done by the lower priority thread.

These two threads communicate by a data structure that represents a *portion* of the execution dependency graph, namely that portion that's able to run immediately (ie kernels with a dependency count of 1). The essence of the patent, and the non-obvious aspects, are how the overall data of interest is split into two parts, essentially a table of kernel attributes and a graph as minimal as possible of dependencies. This dependency graph is further reduced to a subgraph of the parts that might immediately be eligible for execution, to fit into cache on the GPU companion core. The interrupt thread is able to

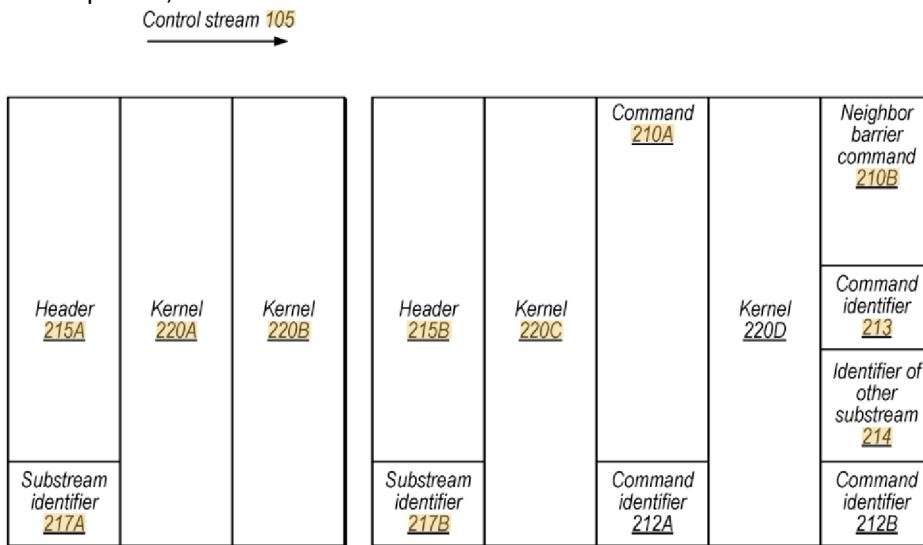
update this portion of the graph with minimal latency, thus establishing the set of newly-executable kernels as rapidly as possible. Meanwhile the lower priority thread will maintain and update the rest of the graph (which, not being in cache, may involve some slow pointer chasing operation through DRAM), including, as appropriate, moving the next set of kernels now with a dependency count of 1, into the cached subgraph.

cross-stream dependency handling

By 2019 we are designing for the future world of M chips, where instead of a single companion core executing firmware to handle all scheduling, some aspects scheduling are going to be delegated to some sort of sub-unit (presumably one sub-unit for an A design, but two or more sub-units for an M [or Pro, or Max] design?)

This has implications for tracking, signaling, and synchronization between these subunits to ensure that kernels maintain execution in dependency order.

Recall, the highest level structure sent to the “GPU” (more precisely the Firmware, according to the earlier patent) is the Control Stream:



As you can see, this contains substreams which, by default, are “absolutely independent” workloads, eg from different processes.

Even so, you may have some reason to want to synchronize two such, apparently independent, substreams with each other, using mechanisms like MTLEvent or MTLSharedEvent.

How can we handle this? Why can't we use the methods described earlier (which synchronize within a

substream)?

What we have described so far is dependencies between kernels encoded in the same substream, with substreams treated as independent by either the driver (old dependency model) or the firmware (new dependency model).

There are clear scaling advantages to being able to split the flow of commands into substreams this way, allowing substreams to be processed independently. In principle we could even process half the substreams on one GPU companion core of an M1 Ultra, and the other half of the substreams on the other GPU companion core of an M1 Ultra.

The patent talks about multiple substream processors, but is unclear as to the scaling. It may have in mind an M1 Ultra situation, or it may already be the case that something like an M1 Max has two GPU companion cores; it's unclear!

It may even be that multiple “Substream Processors” means multiple software structures within a single companion core; each handled on a separate thread, to reduce the overall size of the problem?

Regardless, if we have different substream processors handling different substreams, we need to somehow handle the (admittedly rare) case of cross-stream dependency. This is covered by (2019) <https://patents.google.com/patent/US20200301753A1> *Dependency Scheduling for Control Stream in Parallel Processor*.

In multiple places in these PDFs, I've pointed out that synchronization can be implemented at three levels of sophistication:

- least sophisticated is to stop the world until all pending work has completed, then restart. This is something like a disk `sync()` operation and is, of course, terrible for performance unless that total stop is the exact semantics you require.
- next up is a barrier type solution, where you mark work that's on one side vs the other side of the barrier, and allow through at least some instructions to happen after the barrier (if they are unrelated to the barrier) even while you hold back other instructions behind the barrier, until the barrier task is completed. We've seen, for example, that the M1 CPU uses this a lot for various different memory synchronization operations.
- most sophisticated is a strand type solution where you are able to detect and tag the exact set of dependencies that need to wait in a particular way, and you delay only those tagged dependencies and nothing else.

You can see the barrier solution as delaying by “instruction class and instruction placement in the I-stream”

whereas the strand solution is delaying by “instruction ID, as present in a dependency graph or whatever”.

You can imagine ways that these three schemes could be used for the specific problem of synchronizing dependencies between unrelated GP command substreams; and obviously we want to use the best (ie lowest overhead) mechanism, namely strand ordering.

We can do this because every command in the overall stream of commands has a unique ID, and the

kernel that needs to wait for some other substream has a unique ID, and the substream containing that kernel has a unique ID.

So, in principle at least, it's possible to require that only the affected substream wait, not all substreams; *and* wait only for the specific barrier event of interest, not all barriers or some even larger class of events; *and* require only the affected substream processors, the one generating the barrier, and the one waiting on the barrier, not all processors, to have to engage in the overhead of this process.

So what the patent describes is how (for those circumstances that allow it) the exact dependencies, by Kernel ID and Command ID, are detected in the Control Stream, and strand type dependencies are implemented, forcing as little work as possible to be delayed.

The diagram below, I think makes the point clear.

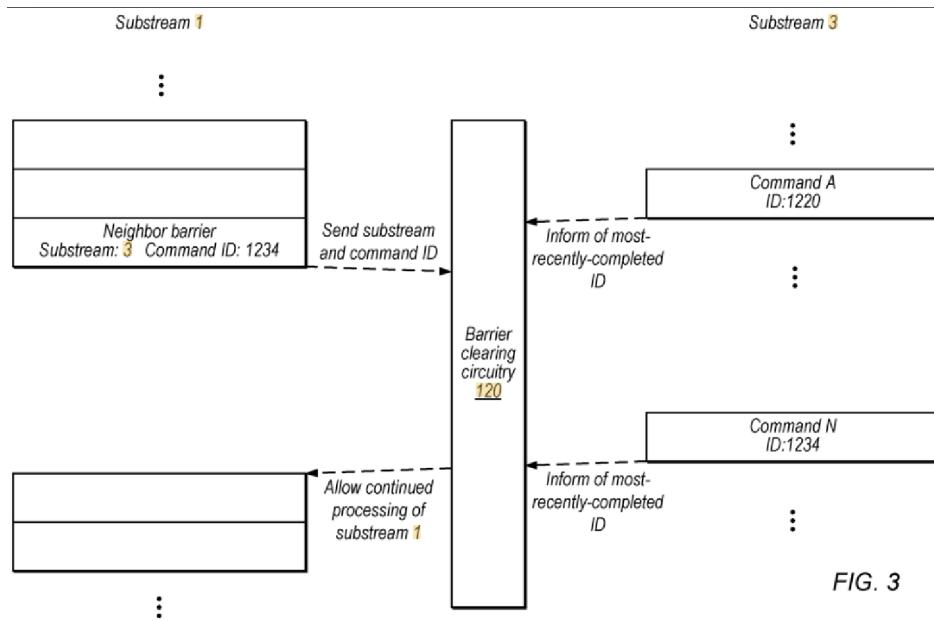
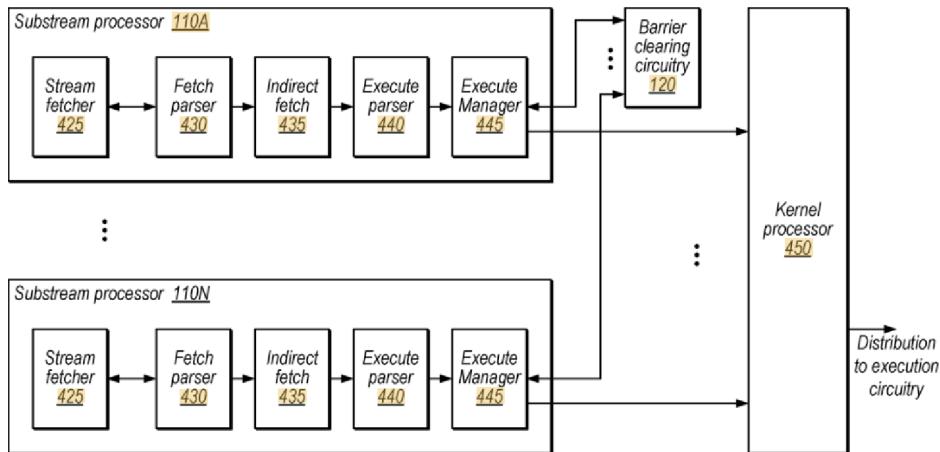


FIG. 3

We also have the diagram below which makes sense in light of what we already learned about Indirect Command Buffer processing:



We see that each of the steps 425, 430, ... 450 are connected by queues, so there is an attempt to perform work at each stage even if other stages are blocked.

So, for example, the Fetch Parser can follow Links (ie redirect the Stream Fetcher to an Indirect Command Buffer, or back from an Indirect Command Buffer to the CPU-generated Command Stream) even if later stages are stalled. This stage happens separately because Stream Fetcher is just a blind “prefetcher” pulling in data; the Parser looks at and interprets that data, including to see that it might be a link.

Next Indirect Fetch retrieves the grid structure in cases where that is indirect (ie stored at some address by an earlier kernel, rather than calculated at compile time), so Indirect Fetch may have to block, waiting first for that earlier kernel to finish, then to retrieve the data from memory (probably L2).

But even while that is happening, Execute Manager 445 and Kernel Processor 450 can find something useful to do in terms of earlier enqueued work.

(There's also, of course, another set of synchronization primitives operating at this level. Most obviously, Stream Fetcher and Fetch Parser must not begin fetching/analysing an indirect command buffer until that indirect command buffer has been completely generated...)

Everything up to Barrier Clearing Circuitry is essentially about detecting and enforcing kernel dependencies; once kernels are past those stages they flow into Kernel Processor, which breaks them up into threadgroups and distributes those across GPU cores, a process we have yet to discuss.

There is more to say on this subject, but first we need to move down one level.

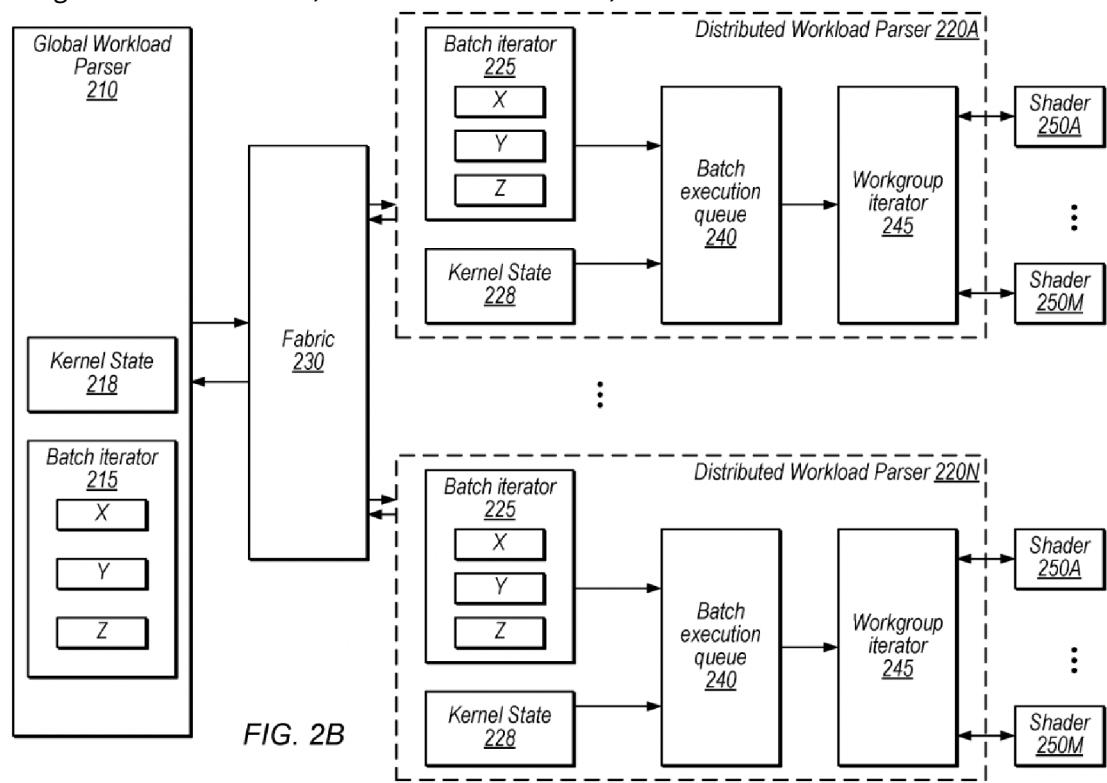
scheduling of batches (distributed scheduling)

We have more thread scheduling in (2018) <https://patents.google.com/patent/US10593094B1> *Distributed compute work parser circuitry using communications fabric*. This is best described, perhaps, as

a set of ideas for building a scalable GPU, once that could grow from phone sized to even multi-chip sized (like an Ultra).

Essentially this patent picks up where the previous patent leaves off, at the “Kernel Processor” labeled block 450.

Imagine that as block 210, Global Workload Parser, below.



The problem to be solved is that we have told Metal about a large grid of work against which we wish to apply a particular compute shader. So ultimately this has to turn into a collection of threadgroups (address of some instruction code, along with various thread IDs, appropriate data pointers, and addresses into the grid).

The simplest solution would be to do this work on the CPU and send the workgroups to the GPU one at a time, but Apple is always trying to offload common-ish work off the CPU.

The next obvious solution might be to do this on the GPU companion core, but that means transfer-

ring a lot of data (details of each threadblock) to each target GPU core.

A better solution is to transfer the work to an intermediary that is close to the core(s) and better able to adapt to each core's changing circumstances.

So we get a hardware solution at two levels.

The first level is a single Global Workload Parser which cuts up the grid into large "batches".

These batches are sent to Distributed Workgroup Parsers, which cut them up into workgroups which are the actual "unit of execution and scheduling".

It's unclear just how many Distributed Workgroup Parsers there are.

One extreme possibility is that there are very few (maybe one on every chip up to a Max, two on an Ultra).

The opposite extreme is that there are one per GPU core. In between, maybe there are something like one for an iPhone chip (so for 4..6 cores) then two for an M chip, four for a Pro chip, eight for a Max chip and so on.

Many Apple patents talk about Compute Data Masters (along with Vertex Data Masters and Fragment Data Masters). In each case the Data Master is essentially dedicated hardware that performs some part of the relevant task. I think, for example, that Distributed Workload Parser is more or less the same thing as Compute Data Master.

If you look at the most recent Imagination Designs, for example, they seem to have one Compute Data Master (and one Vertex and one Fragment data master) for three cores. They also seem to have one Firmware coprocessor (was a MIPS core at one point, is now a RISC-V core) per twelve cores (with a design that scales up to GPU 48 cores).

So it's unclear exactly what the scaling elements are; I think the main takeaway is that Imagination (and Apple, who seems to be following many of their ideas) are continually thinking about scalability - a design that can fit into anything from a phone to a Mac Pro - and part of that scalability is having different work levels (Core, Data Master [eg Distributed Parser], and Firmware) that can all scale up as required.

So the Global Parser sends out batches which are, lets say, maybe 10..30 threadgroups in size. Each Distributed Parser maintains a queue of Batches, and walks these batches generating threadgroups which it feeds to its shaders.

There are queues in the system at two levels, so that even as a Core (ie a "Shader") reports the completion of one threadgroup to its Distributed Parser, it has more work queued up and ready to execute.

Also, as always, maintaining a pool of multiple threadblocks ready to execute means one can engage in heuristics for optimal shading (ie try to match a threadblock against simultaneous execution of a very different sort of work).

Likewise when a Distributed Parser reports to the Global Parser the completion of a batch, it has subsequent batches already queued up, so that no cycles are lost waiting from a reply of new work from the Global Parser.

Using the usual scheme of credits, both levels of Parser know the number of free spaces in their target queues, so can work to keep those queues as full as possible, but not overflowing.

There are a few somewhat interesting details to this setup.

- some degree of load balancing is practised, in that (under normal conditions) each subsequent batch is sent to the Distributed Parser currently with the least occupied Batch queue (and presumably something similar for workgroups from Distributed Parser to Core).
- however some tasks care a lot about data locality and reuse, and it's suboptimal to spread those out randomly. A per-kernel setting (which can be set based on the performance monitoring of the first few threadblocks executed; and perhaps also by the runtime based on compiler heuristics?) can indicate that this kernel is "sequential" in which case the Global Parser makes some effort to direct as many sequential batches as possible to a single Distributed Parser; and likewise sequential threadblocks from the Distributed Parser to a Core.
- the general model is that at the start of batching, a batch size (presumably $x \times y \times z$) is chosen, and batches of that size are handed out. However the system has some flexibility to change that. For example if a target Distributed Parser's queue is close to full, the logic may prioritize "filling up" the Distributed Parser by giving it larger batches, rather than prioritizing Load Balancing (achieved by more, smaller, batches).
- the communication between the Global and Distributed Parsers is somewhat optimized to avoid having to resend redundant information.
- as somewhat orthogonal issues,
 - + the Distributed Parsers also handle communication between the Global Parser and some or all cores, if those cores are forced to required to preempt a long running compute kernel, to allow execution of a graphics or other high priority kernel
 - + and also thrown into the patent are slightly deeper explanations of how the Substream Parsers of the previous patent split the overall task (fetch command streams, while following links and indirect arguments) into multiple queue-separated tasks to allow for prefetching of command-streams in parallel with the parsing and execution of earlier commands in the stream.

construction of batches

We have seen that the Global Parser sends batches to each Distributed Parser. But how are those batches created?

(2018) <https://patents.google.com/patent/US10467724B1> *Fast determination of workgroup batches from multi-dimensional kernels* is a fairly workmanlike patent for how this control machinery decides to create each batch of workgroups from a grid.

Here's the basic problem (which I will describe in 2D, the extension to 3D is obvious).

Suppose I have a grid of size 63×160 . How do I split this up into workgroups (ie the units of work that are individually scheduled?)

The developer may defines a workgroup size (given some constraints, like this must be no more than 1024 elements); otherwise by default this size appears to 32×16 , though it may vary across Apple GPUs. So imagine covering a grid of 63×160 points with a tile of size 32×16 . Pretty obviously we'll get one set of 10 workgroups that are of size 32×16 , and one set of 10 workgroups of size 31×16 . (Apple silicon will

mask out the last lane of the 31×16 workgroup so the developer does not explicitly have to handle this mismatch.)

This is easy enough to understand, but it omits one step.

Imagine covering the grid with the workgroup tile, as I have described above, so we now have this smaller array of 2×10 workgroups. Let's call this array of workgroups, for lack of a better term, a workgrid, so this is a 2×10 workgrid of 20 elements.

The question, then, is how the Global Parser will break up that workgrid into batches of work that are distributed to each core.

First let's discuss batch size. How large is each batch? It's unclear. In the first round of patents Apple suggested a batch size of 8; in some later patents they suggested a batch size of 32. Presumably you want to balance the load balancing flexibility of small batches with the energy-savings (less communication to Distributed Parsers) of large batches. It's unclear if the batch size is dynamic, or where it might be set (Metal Compiler? Driver? GPU Firmware? Based on the current load of the GPU?)

Anyway assume a batch size of 8; then how are batches created? In this first patent, the batch creation is very simple.

Essentially walk along the x -axis first, gathering up elements from the workgrid until you have to wraparound, increase y by one; and keep doing this till you have accumulated 8 elements; that's your batch. Send a description of this batch to the first Distributed Parser, then repeat, constructing the next batch for the next Distributed Parser, and so on. So this algorithm will generate a first batch from our 2×10 workgrid of 2×4 elements, along the x -axis from 0..1, and from $y = 0..3$, then the second batch from $y = 4..7$, and a final batch of only 2×2 elements from $y = 8..9$.

The Local Parser in each GPU core will split these batches into workgroups that are then scheduled as usual.

The *Fast determination of workgroup batches* patent describes some hardware to implement this algorithm so that a new batch can be generated and scheduled to a Distributed Parser every cycle or two.

optimization for more square/cuboid shape (2020)

Now this is a fine start, but is it the optimal way to create batches?

What this algorithm will tend to do (if the grid is large enough) is create long skinny batches that are one unit wide in the y direction and eight units wide in the x direction. But if an algorithm is genuinely 2D-based then a workgroup spread along the x direction probably references some data in the y direction slightly above the locations of the work grid, and this data is probably reused by the next line of workgrids in the y direction. In other words, there is value in trying to structure a grid layout to optimize the amount of data reuse within the various caches.

This argues for, instead of using a batch of 8×1 by default, preferably use say a batch of 4×2 by default (and likewise a batch of $2 \times 2 \times 2$ for 3D). This is the content of (2020) <https://patents.google.com/patent/US20220083377A1> *Compute Kernel Parsing with Limits in one or more Dimensions*, which describes a

slight modification to the circuit used by the 2018 patent so as to give these more square/cube-sized batches rather than long skinny batches.

(At an abstract level this is somewhat analogous to blocking in matrix multiply. And like blocking, it can be done at multiple levels. You could imagine a second level version of this idea that, after taking into account the number of cores available, tries to ensure that the set of 8-sized batches for each Distributed Parser is, as far as possible, a contiguous large cuboid; rather than the current design which feeds multiple *disjoint* batches of size eight to each Distributed Parser. This would give us a second chance at reusing cache data within L1, and a slightly better chance than the current design of reusing data within L2.)

handling of batch completion

Now in the above description I stated that when the Global Parser is given a new kernel, it starts sending batches one by one to the first then second then third Distributed Parsers. That's clearly not going to be sensible if earlier work has already been sent to the relevant Distributed Parsers and thus to GPU cores!

Instead we want to, as far as possible, send each new batch of work to the least loaded GPU cores. Doing this requires machinery, some obvious, some less so.

notification of completion

The obvious piece is that the Global Parser needs to know when each core has completed earlier work. This is one of those patents that only makes sense if you think through ways to solve the problem. We have the Global Parser that wants to know when the kernel is complete.

We have Distributed Parsers that know about their batch(es) of a particular kernel, but not about other Distributed Parsers.

And we have Cores that only know when they have completed a particular threadblock.

The most extreme way to do this is to have each core send a message to the Global Parser each time a workgroup is completed saying that the workgroup of location (x, y, z) is complete. The problem with this is that it generates a lot of chatter, and while it informs the Global Parser of which parts of the kernel are or are not complete, so what? What can the system do with that knowledge?

Next up is have each core send a simple completion message (with a kernelID) to the Global Parser each time a workgroup is completed. When the count of completions equals the count of thread-groups, we're done. This still generates a lot of chatter, and again provides more info (a fine-grained knowledge of completion rates) that's more than needed. It also requires a path from each GPU core to the Global Parser.

So let's exploit the idea of batching that we are already using. Each core reports completion of a threadblock (of some kernelID) to the Distributed Parser. The Distributed Parser counts these up and at some granularity (more or less matching the granularity of batches, so after say 8 or so threadblocks com-

pleted) sends a single message reporting this fact to the Global Parser.

In fact the aggregator reports to the Global Parser “every so often” the aggregate results (ie so many workgroups of kernelID have been completed, at what seems to be a combination of when the count gets high enough, or when the count has not been reported for some time. This second case is an easy way to handle when we’re approaching the end of a kernel, without the Distributed Parser otherwise having itself to track the number of outstanding threadblocks. (If the last Batch you received was only 3 threadblocks, you’ll be waiting forever if you only report once you get 8 threadblock completions for this kernel...)

This allows the Global Parser a “good enough” view of which work has been completed vs which work is outstanding, without requiring too many messages.

This is described in (2020) <https://patents.google.com/patent/US20210279832A1> *Completion Signaling Techniques in Distributed Processor*. The ideas are simple; the bulk of the patent is how to implement this in logic that uses minimal energy. For example when the Distributed Parser receives a completion from a Core, it needs to look up the relevant kernelID to be able to increment the completion count for that kernelID. How to do this? One solution is a CAM but that’s a lot of energy; alternately hashing and an SRAM, but that’s a lot of logic and probably overkill. Apple uses a FIFO ordered by the kernels as they are given to the Distributed Parser. They don’t say so, but my guess is the FIFO is probed sequentially starting at the end, and the common case is that usually only three or four kernels are actively running, so only a few sequential tests are required to match the kernelID of interest.

avoiding priority inversion

The above technique gives us one way to track what’s happening across the various cores, and so one degree of insight into which cores are more lightly loaded. But there are other elements to optimal scheduling. In particular we want to avoid full queue blocking, ie situations where lightweight (but high priority) work is unable even to be queued (and thus arbitrated as high priority) because every queue slot is occupied by expensive work. In such a situation, the most aggressive way to deal with this is by forcing preemption, but we’d prefer alternative, less expensive, solutions, as much as possible.

How can we prevent queues filling up this way?

One part is you need to know something of the complexity of a kernel. This is solved by marking kernels with a value that indicates how “complex” they are; at least an approximation to how long a threadblock will take to execute. This value is set by the Metal Compiler. Even if you are assembling an indirect command buffer, I think ultimately you land up pointing to instructions that have previously been compiled by the Metal Compiler, so this tagging occurs along all Command Buffer construction paths.

Ideally you’d augment this with whatever you learned from performance monitoring, so that in principle the kernel complexity might change as it is executed and we learn (eg as a result of operating on unexpectedly difficult data) that it’s a more complex kernel than expected.

Suppose you know a kernel is complex. Then what?

One (not great, but maybe better than nothing) solution is to send out smaller batches for complex

kernels, so that less work is queued up in any particular Distributed Parser (which will have some limited storage, say four or six entries, for Kernel Batches). But sending many smaller batches costs energy, and the whole point of the batching scheme is to reduce energy.

What Apple does instead is send out fewer batches, so that instead of the Distributed Parser queuing up say four batches of ComplexKernel, it only queues up two batches, so that the worst case slowdown any new high priority LightKernel might see is a wait till two batches are processed, rather than a wait till four batches are processed.

The same number of batches ultimately are sent out, so we don't pay a higher energy cost.

This is attacking the root of the problem – we don't want all queue slots occupied by complex kernel batches.

By definition, we do now use less of the queue, but that's not a problem! The point of the queue is to provide work even if there are delays in the Fabric or the Companion Core, the Distributed Parser always has something to work on. But if the kernels really are complex, then two queue entries will provide as much “queued up work” as four light kernel batches, so using less of the queue is not a problem.

This same scheme recurses down to the Cores. The Distributed Parser likewise will queue up fewer threadblocks of a complex kernel on a core so that, under most conditions, the work in progress on the machine will be able to drain fairly rapidly if we wish to do.

The rest of the patent then is details about how this is implemented via each higher level system (Global Parser or Distributed Parser) knowing (via credits) how many queue entries are free in each of the devices below them, and, for complex kernels, reducing the number of credits so that these kernels effectively are allowed to occupy fewer slots in these queues.

Now when a lightweight kernel arrives, it can be distributed to each Distributed Parser, and then its threadblocks on to each Core. There will be empty slots in the queues of each of these (because slots were deliberately left empty) and then standard QoS arbitration should ensure that these lightweight kernels will, if appropriate, be first to be converted from batches to threadblocks, then have their threadblocks able to be enqueued on each Core, then first to be scheduled once an earlier threadblock completes.

This is all fairly easy to implement given the underlying technology already present in the GPU, but does a nice job of improving load-balancing and limiting latency behind complex kernels.

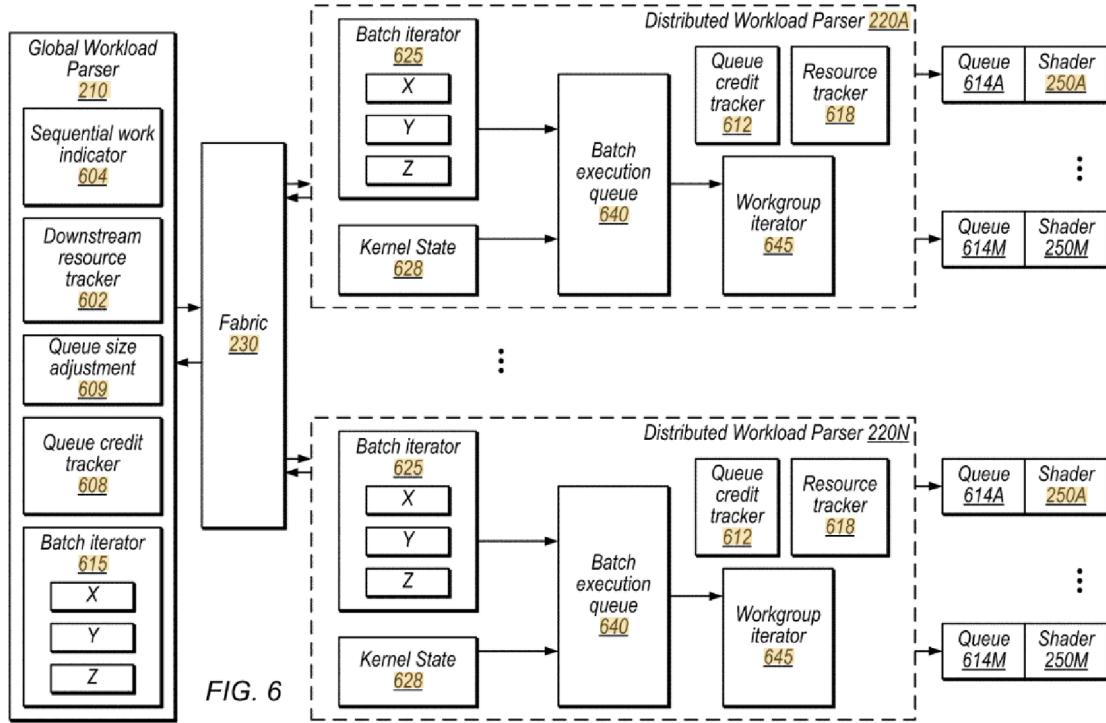
(2020) <https://patents.google.com/patent/US11500692B2> *Dynamic buffering control for compute work distribution.*

At this point you can understand this diagram from various of these patents, including eg the above <https://patents.google.com/patent/US11500692B2>.

We've discussed the relationship between 210 (Global Parser) and 220 (Local Parser), the Batch Iterator 615 and how it sends batch descriptions (625) to each Local Parser, the Queue Credit tracking scheme 608, and how it adjusts the effective size of the Buffer 640 so that more complex kernels don't crowd out lighterweight kernels. Workgroup iterator 645 splits each queued batch into workgroups (of maxi-

mum size 1024 lanes) and queues them in queues 614 which schedule them on shaders as execution slots become available.

Sequential Work Indicator 604 attempts to ensure that kernels which strongly benefit from cache locality have a succession of neighboring batches all sent to a single core. We saw that in a subsequent patent that the effectiveness of this is somewhat improved (made 2D or 3D aware) by creating batches that are close to square to cube shaped rather than long and skinny.



more optimized batch sizes

We're going to see in 2021 a grand rationalization of everything to do with high-level scheduling, but before then let's consider one last tweak to the system. We've assumed that batches are more or less identically sized, and that's pretty much all the scheduler can assume for compute kernels. But graphics kernels have a known structure (based on tiling across the screen) and a repeated structure (frame after frame, most frames like the previous frame). This gives us useful material to work with!

Ways to utilize these facts are given in (2020) <https://patents.google.com/patent/US20210287324A1>

Graphics Processing Techniques based on Frame Portion Cost Estimates. What this patent says is that, as each frame is generated, statistics for groups of tiles are gathered based on things like how many polygons intersect this group of tiles, how many polygons require blending, texture complexity, and so on.

So, suppose that we execute frame A, and now have just started to execute frame B, so we have some estimated costs for frame B but have not done any further work. (Estimates can be of two forms, some late stage estimates based on frame A, some early frame estimates based on already processed parts of frame B like tessellation/geometry processing and tile-binning.)

How can we use these statistics?

- The first possibility is that some of the early stage *estimated* costs for an area of the screen (ie a sequence of groups of tiles) for frame A match the estimated costs for frame B. With some experience, and given the relative size of the groups of tiles and the number of successive statistics matches, we can assume that this is vanishingly unlikely except in the case that the two frames (or at least *this portion* of the two frames) are the same and so can save energy by not redrawing the frame. (Obviously this will not work if we have a static geometry but change our textures or lights! The estimated cost statistics we are tracking for each frame also have to include all these elements in some fashion, eg by hashes of the address of textures or whatever.)

- The second possibility is that suppose the estimated costs for frame A are close to the estimated costs for frame B. Then we can assume the later stage tasks for frame A and frame B are of comparable complexity, and can replace the *estimated* costs for frame B with the *measured* costs for frame A, which in turn will give us a better estimate for things like

- + the frequency at which we need to run the GPU to generate the frame in time for presentation but with minimum energy cost.

- + load balancing the more expensive regions of the screen against the less expensive regions. All of this might seem like overkill for traditional polygon based GPUs, but you can easily see the value when you consider more sophisticated graphics (eg ray tracing, as already mentioned, or regions of particle-based imagery).

Load balancing can be achieved after the fact either by work stealing or by preemption, but if we can avoid those by optimal scheduling of tile batches, we're better off.

- A third way we can use these estimates is if a situation arises where preemption may be required. We can look at the deadlines the preemption is supposed to meet, and the time estimates of both the high priority task and (if it is also a graphics task) the low priority task, and draw conclusions about perhaps we can get away with not forcing a preemption.

rationalization of scheduling- 2021

We have seen repeatedly that Apple is not shy about ripping up a pre-existing subsystem and replacing it with a very different design.

This is what we get in 2021 with a new GPU Scheduler that tries to do everything we have already seen, but in a more rational and scalable way, with (2021) <https://patents.google.com/patent/US20230047481A1> *Affinity-based Graphics Scheduling*.

As before we start with Firmware being given a stream of kicks, and handling dependencies/synchronization between Kicks.

The runnable kicks (ie those for which dependencies are satisfied) are passed, together with the pre- and post-commands (stuff like flushing/invalidating caches) on to a *Kickslot Manager* which decides how to distribute them. Distribution means to which cores to send a kick (ie a kernel); and it's important to realize that scalability in this context means we're now thinking of Ultra type devices. In fact the patent talks in terms of a more extreme device with, say, four sets of GPU hardware on different chiplets.

Remember that on an Ultra device we have cores with their local L1D sharing a common L2; but we have two L2's each with nearby GPUs. The two L2's are coherent with each other, but it's obviously power and performance optimal to keep threadblocks that are communicating with each other sharing the same L2, rather than constantly having to move lines between the L2's.

Apple suggests three distribution modes that may be of interest:

First is a large kernel with very little, and mostly local, data sharing. There's no reason not to spread this over both sides of an Ultra.

Second is a kernel that engages in a lot of data sharing, and it's usually optimal to place this on one chiplet, ie interacting with only one portion of the L2, so that L2 lines don't have to move between chiplets.

Finally there is the small kernel that should run on a single core. This might be because the task is easily localized in a way where this makes sense. For example suppose we have reached the fragment shading stage of the graphics pipeline, and we have say a ten-core GPU. The Apple graphics software could split the target window into say eight super-tiles, and create a fragmentation/pixel shader kick for each super-tile, then run each super-tile on a separate core. This would give good data reuse of both L1D and textures (since some triangles straddle two tiles) and is easily implemented.

A fourth possibility, which Apple does not suggest, is if they want to follow nVidia and support a thread grouping option larger than the threadblock; in that case you might want to pin a kernel to a particular subset of cores, say four particular cores that are tied together in a way that they can somewhat share their Scratchpad storage and provide fast synchronization.

So how do we know how to distribute a kick?

Kicks come in with various flags attached to them, which includes “initial suggestion” distribution flags. These flags can be overridden by the distribution software depending on the current distribution of work. For example if a task suggests that it's best limited to a single L2, but there is no other work going on, we might as well split it across both halves of an Ultra and enjoy some speedup! On the other hand, if there is plenty of work already queued up, then we might as well honor the single L2 distribution request, and then shift later work to the other side of the Ultra.

So now that we know the task (distribute shaders while honoring their distribution [ie cache affinity] requests) how to perform this?

The primary work is done in what we used to call the Global Parser (and which is now called the Pri-

mary Control Circuit).

Recall that this used to walk through a shader generating batches, then would send out those batches (usually in a kinda Round Robin fashion) with some slight tweaks either to send successive batches to the same core (a simple attempt at affinity) or to limit the sending out of complex batches relative to lightweight batch. What did this scheme do when there were no queue spaces available anymore in any Distributed Parser to accept more Batches? it's unclear; perhaps it just halted until a queue slot opened up? That certainly limits, for example, the ability of a later lightweight kernel to move forward until all the batches of an earlier kernel have been enqueued and sent out.

The new scheme provides a scalable version of these tasks, for example, deals with the problem described above.

We define multiple (for example four) pipelines that perform the above set of tasks. Conceptually a shader comes in and gets placed on one of these pipelines. Whenever a particular pipeline is blocked in a task, the others can keep going. These don't have to be physical hardware pipelines (though they may be); they could simply be multiple separate processes executing on the Companion Core.

The second is that we

The diagram below looks complicated but it's basically the revised version of the Global Parser. Start at the bottom rather than the top. Each pipeline generates Batches of threadgroups from its Shader, as we have seen before. Each batch is assigned to a particular mGPU (as before this term is unclear, but think of it as a Distributed Parser managing say 3 or 4 Cores). Since each cycle multiple pipelines may generate a Batch, we need an Arbiter to decide which pipeline wins, and the relevant batch is sent to the target mGPU/Distributed Parser.

But what determines the mGPU to which the batch is sent? In the past that would essentially have been established by round robin across the Distributed Parsers (modulated by the complexity and sequential work issues we've discussed). In the new model, this is done by the mGPU assigners (650) which make the decision based on a mask of allowed options. So if every bit in the mask is set (650) will choose the least loaded GPU core for assignment; if the left half of the bits are set then only the least loaded GPU core on one of the two Ultra GPUs, and if only one bit is set then there is no choice.

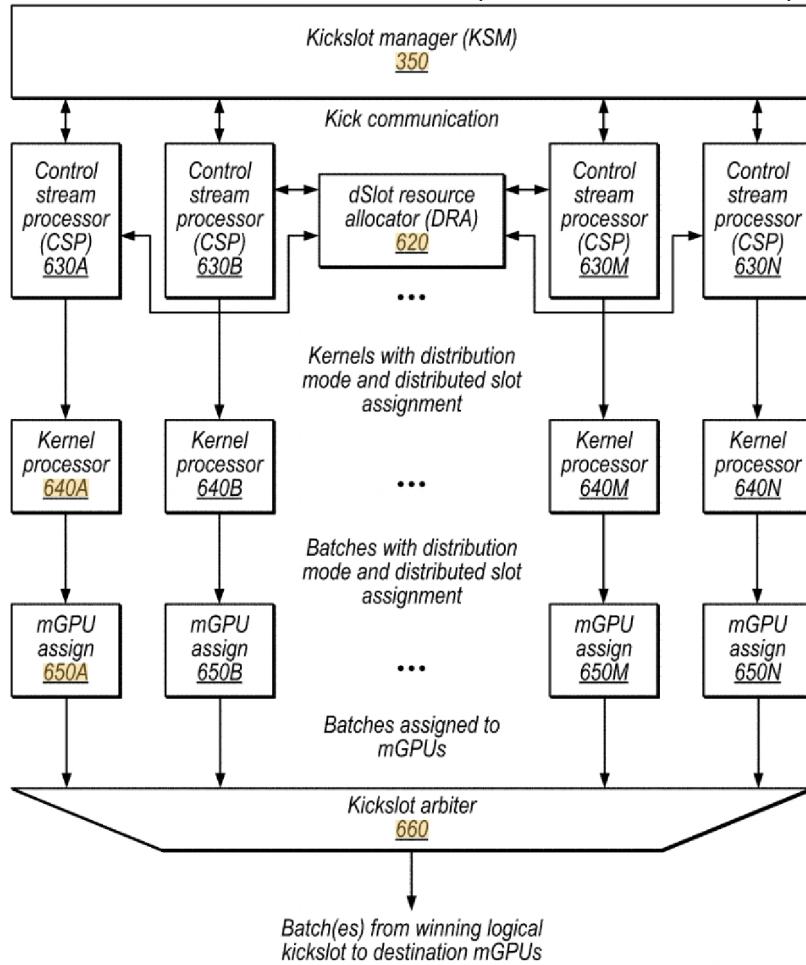
Above (650), (640) are the familiar Batch Generators that carve the next Batch of threadgroups from the kernel grid; that answers where the Batch for 650 comes from, but what about the mask?

That comes from 620, the dSlot Resource Allocator, which looks at the requested distribution mode, possibly overrides it, and then decides on an appropriate mask based on how busy the GPU appears to be. For example if the distribution mode is limited to one physical L2, then (620) will choose the chiplet that currently appears to have the least work queued up on it. You can think of the resource being handed out here as somewhat analogous to the queue slots in Distributed Parsers, and the DRA may hold back some of these queue slots for the same reasons we've already seen, to ensure that queue slots are available for lightweight high priority kernels. But this scheme is slightly more flexible in that rather than reserving n queue slots on each Distributed Parser, we're reserving n possible queue slots across the entire GPU, to be allocated later as necessary.

Finally before the kernels are Batched and sent out, their pre-commands and the fetching and

following of links and indirect addresses, and then later their post-commands, are executed by the CSPs (630). In other words the CSPs perform the task that was handled by what used to be called Substream Processors as of the 2019 patent, and this will include the rare, but necessary, occasional waiting on a cross-stream barrier, as described when we talked about Substream Processors.

This CSP overhead can be done in parallel with the decision process of the DRA.



One important point is that scheduling is now deferred to as late as possible. The usual case is that the mask of possible GPU cores has more than one entry, and we only decide on the actual target core at the last possible moment, which allows for better load balancing.

In a way we have now split the large-scale scheduling task into two parts.

The first part breaks kicks up into batches assuming an abstract machine of unlimited size; it's concerned with the relative ordering of kernels, and with breaking up kernels into batches. But not with where to place batches.

The second, separate, step then decides which is the optimal GPU core to execute each successive batch. Since this step can occur very close to the point at which the batch is sent out to begin execution, it can do a better job of load balancing.

With all this background, then, the main point of the patent is ways to allocate kernels to optimize for affinity as much as possible. The algorithm for assigning a batch to a core looks something like:

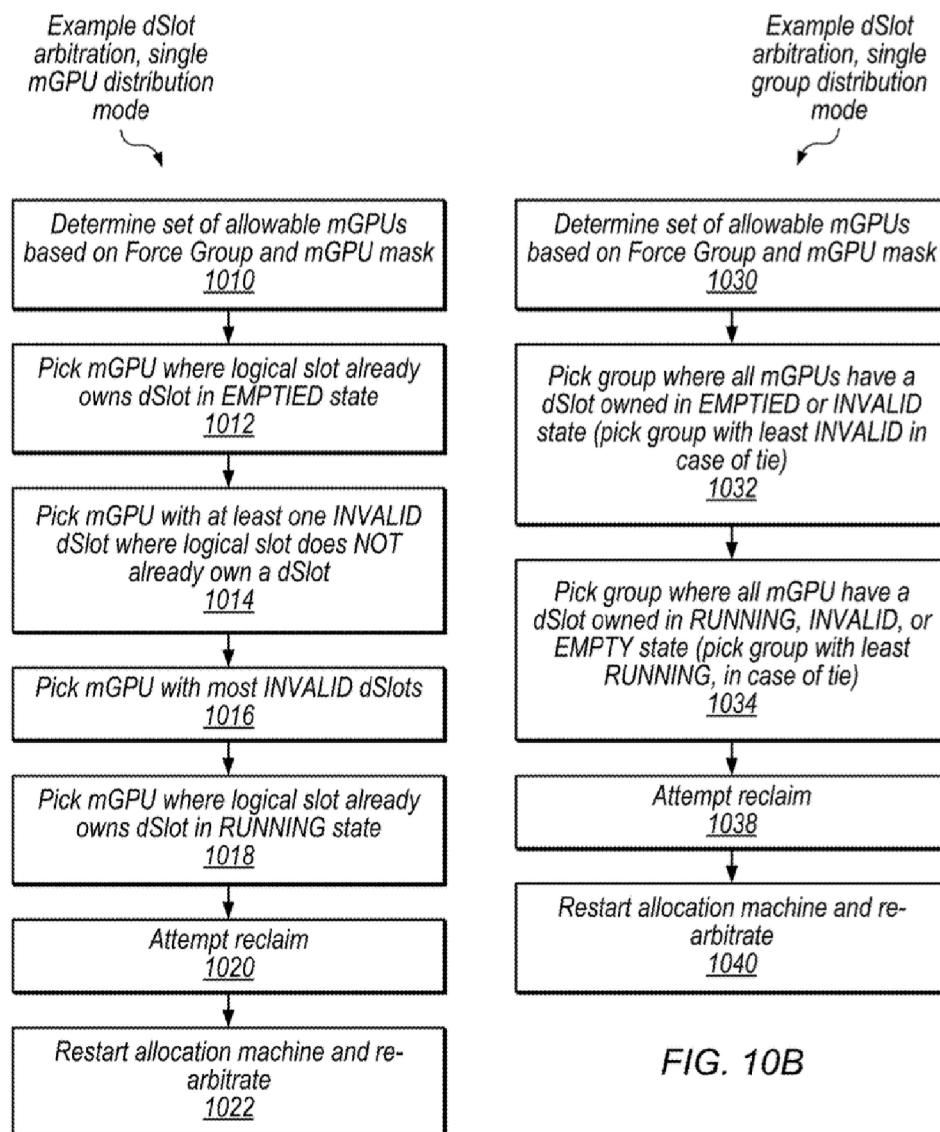


FIG. 10A

FIG. 10B

Even without understanding the details, it's clear that we are trying to run kernels

- firstly on cores that previously ran the same kernel OR
- secondly on cores that are now invalid (finished with everything to do with a particular kick, so cache affinity no longer matters)

Once a GPU core has been configured for a particular shader and has then completed its work, the core does not revert to an INVALID state, instead it reverts to an EMPTY state which indicates that it is still “configured” for the particular shader; and if possible we use such a core(s).

Next choice is an INVALID core, so that we're not stealing a configured core from another shader.

If we have to, we'll add our work to already configured core (which means, of course, that we'll have to share execution resources and cache).

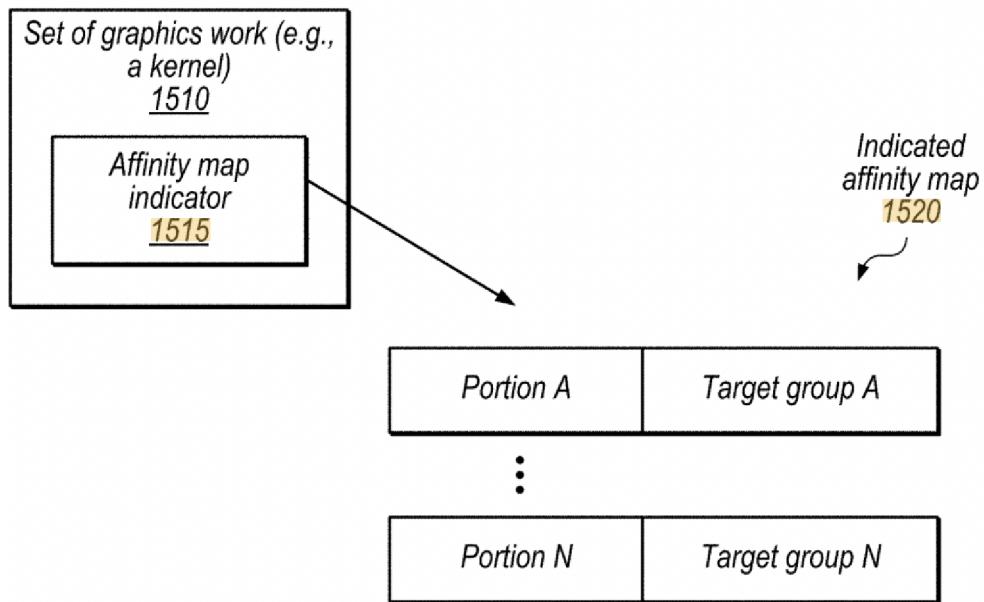
In the absolute worst case (simply no cores available with free space) we might force a Reclaim. This is only done for high priority shaders.

The Reclaim process first looks for cores that are in a Flushing state (so moving data between caches, where we will have to wait until the flushing is done before starting the kernel), and, in the worst case scenario, with important enough priority, may force an existing core to context switch to a more pressing kernel.

Another feature of the previous design was an attempt to “reserve” some queue slots within each GPU core for high priority kernels, to try to limit the frequency of forced preemptions, and that same idea is retained, though implemented slightly differently.

(As something of a side issue, there's also one page that talks about a way to optimize flushing between kicks. The standard way to perform this flushing was scoped to the core(s) that did the work, or perhaps to the entire GPU. It's now feasible and sensible to introduce a third scope where the caches to be flushed are on one particular Group of mGPUs, ie are on one particular chiplet, since there will be [given how Ultra's are built out of Max's] some easy way to flush the subset of caches that are limited to one particular chiplet, and only this subset needs to be flushed if all kernels of a kick were restricted to this chiplet.)

After all this build up, the main event is in Figure 15.



Between scheduling of kernels and splitting a kernel into batches, we add a stage which you can think of either as creating sub-grids, or as creating super-batches. Either way we split the grid of work by the number of chiplets (so split the grid in half for an Ultra) and send each half to a separate Batch Generator (probably located on the relevant chiplet).

The hope of this exercise is that each half grid threadblock will mainly access data used by other threadblocks in the same half-grid, thus in the same L2, so we get better cache affinity.

Overall this is more or less the obvious way to split work across any multi-GPU system, though it's easier for an Ultra because the L2's are coherent (so things will work even if data is in the "wrong" cache) and communication between the L2's is efficient, so we lose some energy crossing between L2's, but not too much time.

The patent also makes the geometric point we've already seen that, if you only have two chiplets, the only sensible choice is splitting the grid along the x or y or z axes, if you have four chiplets it makes sense to try to generate four "square-ish" sized sub-grids, rather than say four long-and-thin sub-grids.

Obviously strict splitting of a grid as described above may be sub-optimal if one half of the grid completes substantially before the other (eg we're ray tracing and the top half of the image is much more complex than the lower half). So there's also a work-stealing mechanism in place; once one of the chiplets finishes work, batches from other chiplets can be sent to that chiplet. L2 coherency will ensure this works, though it will require moving some data between the L2 caches of the different chiplets.

The patent ends by discussing the extent of pipelining/overlap the kernel scheduling strives for. The flowchart below describes the evolution of a “tracking slot” which is essentially a storage slot holding the state of one of the various kernels the firmware knows about. At any given time the firmware is aware of multiple kernels in various stages of execution, and there’s an attempt to do as much “overhead” work as possible in parallel with other kernel execution.

Some of the stages below are understandable. For example RCE is Register Copy Engine; this loads various settings that will be required to launch the kernel and stores them somewhere “more rapidly accessible”. I think that, realistically, it’s loading them from DRAM or SLC, and storing them in L1 cache of the GPU companion core.

Once that parameter copying is done, we wait for parent kernels (ie kernels we’re dependent on) to complete, before being considered for actual scheduling.

I think “Wait for resources” means wait for an execution slot in one of the vertical pipes of Control Stream Processor+Kernel Processor that we saw earlier. Then we wait for allocation to a GPU core (maybe one core, maybe one chiplet, maybe multiple chiplets).

At any stage, as you can see, the process can be halted, eg because a higher priority kernel has arrived. At the early stages (or at the very last stages when the kernel has completed) halting is easy, otherwise it requires more complicated unwinding,saving of state, and reallocation of resources.

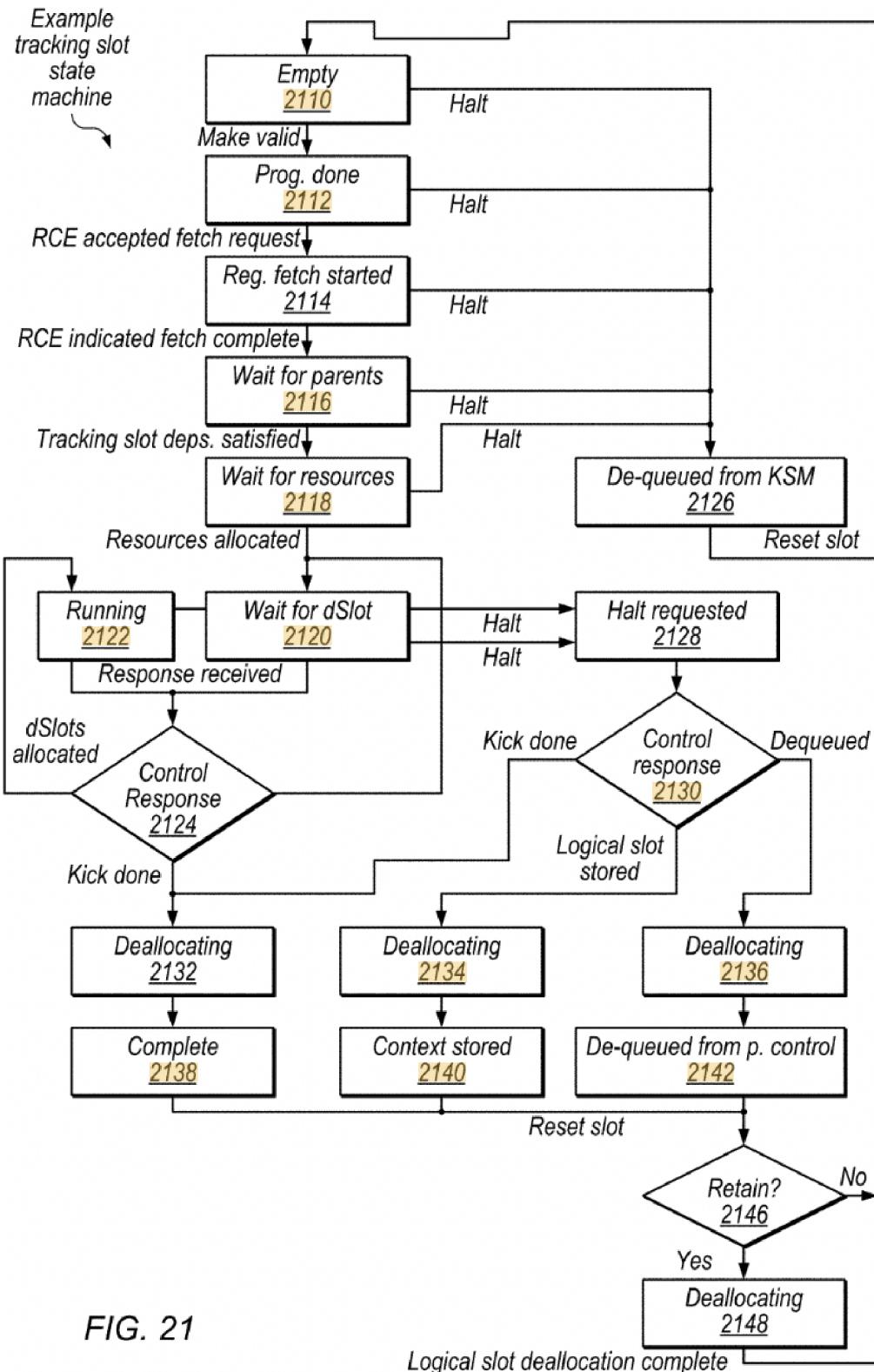


FIG. 21

territory, but adds a whole new section at the end. The new section describes performance registers that are effectively per kernel (technically they are per command buffer, but a sequence of kernels in a command buffer is serialized by definition). These registers collect data in each GPU core and report it back to the firmware.

This then is used as expected

- to try to maintain QoS (which mostly means each kernel/command buffer getting appropriate fraction of the available throughput), enforced by throttling the rate of batches submitted by kernels that are grabbing more than their fair share
- to detect priority inversions (high priority kernel either blocked by, or dependent on, a low priority kernel) and boost priorities of the blocking kernel, or even preempt it, so that the high priority kernel can execute ASAP
- (presumably, based on statements in other patents) to try to balance the types of kernels sent to each GPU core so that, when possible, a mix of work types utilizes all the different hardware in a GPU

There are a variety of additional small tweaks scattered throughout the patent.

For example in some cases (I assume things like multiple small kicks back to back) the patent says there's no need for an interrupt (to let the SW side of things know progress), it's good enough to submit an interrupt after all the kicks are done. So there is a per-kick setting whether or not to generate an interrupt when done.

Likewise when distributing various setup data, the targets may be multiple to all cores. Rather than re-sending the data multiple times, it is sent as a masked broadcast (ie broadcast with a mask to indicate the targets). Even on a point-to-point network, like a ring (rather than a bus) this still saves power and time because each core can read what it needs, if anything, from the broadcast, then forward to the next core in the ring.

Likewise setup registers for the various pieces of hardware of the system are laid out in “order of use”, so that even while kick N is being configured, or executing, on any of these pieces of hardware at different levels of the system, the setup registers for kick $N + 1$ can be set ahead to override the earlier (and no longer necessary) setup registers of kick N . Some registers for high priority kicks may even be forwarded and configured before it is absolutely certain that this high priority kick will be the next to be executed [in principle an even higher priority kick may appear while we wait]; this “prefetching” can reduce latency until the high priority kick begins.

future of scheduling

Now this basic design I have described (specify dependencies in the Metal API, schedule kicks in a particular way, do sync work between kicks) sounds good. But there is evidence that it is reaching its limitations and at some point Apple may update it to something rather different (probably initially via HW changes with the Metal SW inserting code to match a new synchronization model, then later a change to the Metal API, perhaps when Metal drops non-Apple-silicon support).

To see the problem, imagine, for example, a very simple kernel consisting of vertex processing then

fragment shading. This will form one kick (equal to a frame) and a given kick may well have dependencies on the previous kick (if for no other reason than the pixels of the later frame needs to execute after the earlier frame!)

But think about this at a more abstract level. The point to remember is that

- the unit of ordering and dependency tracking is the kick. A kick is something that doesn't have any internal assumptions about memory visibility, so, essentially, between kicks is when L1D is flushed out to L2 and becomes visible to other GPU cores.
- while for compute purposes a kick is (more or less) the same as a compute kernel, graphics tries to be more aggressive, and to pack multiple graphics "kernels" into a single kick. Graphics can get away with this because the graphics system can, if necessary, add additional synchronizations or L1 cache control calls in a way that's not inherent in the Metal dependency model.

So return to our assumption of the simplest possible kick consisting of a vertex stage and a fragment stage.

The vertex processing of the second frame probably has no dependencies on the earlier frame, and uses different hardware from the later stages (the fragment processing stages) of the later frame. If there were a way to specify more precisely the dependency relationships between *kernels* (rather than between *ticks*) you might be able to launch vertex processing for the second frame while the first frame was still performing fragment processing. This has multiple advantages:

- you increase performance because now you're using your vertex hardware in parallel with your fragment hardware rather than using them sequentially
- you reduce energy because all the overhead costs (keeping the GPU as a whole, the caches, the register file etc) alive are shared across these two stages.

These claims mostly still hold true even if our vertex stage includes a geometry shader, and likewise fragment processing includes a pixel shader; even though now there will be some hardware overlap when both wish to use the shader hardware, they will be rather different types of code, probably allowing for good overlap in terms of things like making different use of DRAM bandwidth, register and operand bandwidth, the texture unit, and so on.

Right now the design of the system doesn't optimally allow us to achieve this. Specifically because of the way dependencies are specified and tracked, we can't fire off the vertex part of the second frame kick if the fragment part of the kick depends on, say, something created by the earlier kick; the kick is atomic in scheduling, and we can't, *within the model*, break it open and schedule the vertex part immediately while delaying the fragment part until the dependency rules allow it.

But we can try to get this result (for this specialized case; obviously we'd prefer a more generic solution longterm) via a suboptimal route. Look at (2020) <https://patents.google.com/patent/US11055812B1> *Opportunistic launch of idempotent geometry stage render operations*. This patent describes essentially the problem I have presented, with (as a special case) the firmware looking to see if it appears that the vertex processing for the next frame can be launched within the fragment shader portion of the kick of the current frame. If that's possible, then why not launch vertex processing early? The problem they see as being most likely is that "memory runs out". I think what they mean by this is

local address space, if the usage by the new vertex stage plus the usage by the previous fragment stage exceeds local address space capacity. Especially if this is a hack being added onto the system via tweaking the firmware, the paths may not be available to legitimately check for resource allocation exactly the way the system was designed.

Of course, in the event of resources running out, the previous fragment stage has to win, and we simply kill the vertex stage. Which in turn, of course, *also* means that we have to be careful that the vertex processing is idempotent, that is whatever it does doesn't have side effects beyond what will happen anyway the second time the vertex processor executes (at the end of the previous frame).

So one can see this as just a quick hack (my guess is done via a firmware update) to reduce frame processing latency a little; but it also suggests that Apple is aware of the suboptimal aspects of the current dependency model (from Metal programming down through firmware to hardware) and is probably going to revise it at some point.

It's very easy to think that kernel and threadblock scheduling are not that important because they don't happen that often, relative to warp scheduling. But things like false dependencies between kernels can delay a kernel for a lot longer than it takes to actually execute the kernel...

Instruction Flow

general comments about the instruction set

What can we say about the instruction encoding?

Dougall's work at <https://dougallj.github.io/applegpu/docs.html> tells us that instructions are variable length, and one thing that's clear is that high bits of register specifiers are placed in the optional extension bytes to an instruction so that in the (common, I guess) case where not all registers are used those extension bytes can maybe be omitted.

However (2013) <https://patents.google.com/patent/US9378146B2> *Instruction source specification* describes registers specified in a different way, via a "mapping" table. The idea is that the mapping table (appended to the end of the instruction) has one to four entries, each a register ID, and the instruction itself says that operand A comes from table entry 1, operand B from table entry 2, destination to table entry 2, etc. This allows cheap identification of where registers are reused or overwritten. It's unclear if this idea was abandoned, or is used by instruction rewriting during pre-decode?

The actual instruction set is somewhat like ARMv8, and like other GPU instruction sets, in that instructions generally have multiple "modifier" bits to pack extra work into a single instruction; for example bits to negate, take the absolute value, or saturate.

We will eventually discuss the register cache, but each instruction also has several register cache control bits offering rather more control than other GPU ISAs.

Another uncommon feature is that instructions can indicate a conversion between FP16 and FP32 within many instructions, and this type conversion is handled within the data path, so that type conversions are essentially free, bundled along with the "real" operation.

The current instruction set is primarily scalar (as opposed to the PowerVR ISA of the A7); that is each lane of the GPU executes the equivalent of CPU scalar instructions on scalar registers, as opposed to say each lane executing SIMD instructions on SIMD registers.

Apple calls the loads and stores “vector” loads and stores, but I think it’s probably more helpful to view them as the equivalent of the CPU load pair or store pair, with the option to extend to wider than pairs, up to at least quads of vectors.

Just like CPUs there is a select instruction, and just like CPUs, it’s preferable to use it for appropriate cases.

instruction flow - the big picture

Apple’s instruction flow is not obvious, so, as we have done with other complex and unusual ideas, we’ll describe it once as an overview, then in much more detail.

The actual code fed to each core consists of instructions which are grouped into “clauses”.

Clauses have at least two roles

- they are atomic execution units, so once instruction of a clause begins, it persists till the end of the clause
- they are units of caching (so you can think of them as “macros” or as what some systems call “millicode”, code blocks that are smaller than a function but which can be accessed as a unit, or even as multi-cycle “super-instructions”).

To this end, clauses cannot contain control flow.

(Or more precisely, control flow lives in special single-instruction clauses. Specifically these are not if/then instructions, handled by predication; rather control flow instructions are instructions that change the PC, ie jump to another clause, or loop back to the previous clause.)

Going upward clauses are grouped into “streams” and streams are grouped into “kicks”. We’ve seen a whole lot of work on how kicks are handled, things like how firmware inserts the cache flushing instructions between kicks, and how dependencies between kicks are tracked so that pending kicks can be held until all their dependencies are resolved and they become runnable.

Clauses likewise have dependencies – within a clause one instruction may depend on the result of an earlier instruction – and across clauses, so that clause B depends on a result generated by clause A.

Obviously these dependencies have to be tracked and honored.

In a modern OoO CPU various aspects of this dependency tracking takes up most of the area and power, because we want to wait for the absolute least time (zero cycles if possible) that dependencies enforce, and are willing to build the machinery of speculation to even bypass dependencies when we can get away with it.

In the GPU core we do things rather more like we tracked kick dependencies. *Within* a clause we might note that a result will be available after 8 cycles (ie that’s how long the operation takes) so we might flag an instruction as having to wait at least 8 cycles. This is somewhat like an in-order CPU, except that we will try to execute instructions from the clause of another warp while we wait.

Between clauses we track dependency at a coarse granularity, something like “clause B can’t execute until clause A is complete”. We don’t bother with fine-grained tracking of which instruction within clause B depends on which instruction within clause A. This allows for much less tracking overhead, and mostly works out fine because we just execute other warps while clause B waits for clause A to complete.

For instruction storage there is the L2 cache, a per-core L1-I cache, and multiple L0-I caches in front of different execution units of the pipeline (eg the sample unit, the interpolate unit [together used for texture lookup], the load unit, etc). Each clause is not just a run of instructions between control flow, it’s a run of a particular type of instruction. Thus a run of load instructions might form a load clause, a run of store instructions form a store clause, likewise a texture clause or an arithmetic clause. Apple tends to call arithmetic [which is mostly FP but with the occasional integer instruction] “datapath”. The L0I caches in front of different units store a few clauses of the instruction type appropriate to that execution unit. Execution consists of simply walking the clause instructions sequentially (much cheaper than accessing a cache), and occasionally switching from one clause to another to honor dependencies.

AMD’s newest (CDNA3) designs have an interesting trick where instead of a 32kB L1I cache per core, they use a 64kB L1I shared by two cores. Since so many shaders run simultaneously on multiple cores, This mostly feels like each core has a 64kB L1I. Probably worth considering for an Apple design!

Another idea that could be used by Apple is that, as we will see, Apple’s current designs have each core communicating with other cores as a chain, each core linked to a left and right neighbor, with messages passed along the chain till they eventually reach the target (eg the L2 cache). You could imagine requests to the L2 cache could, at each stop along the way, also look into the intermediate node’s L1I and L1D to see if the data is available. If we did this for every node on the way to L2, and waited for the results before passing the request on, the L2 latency might be too high. But you could imagine variants (separate the cache lookup from passing the request on? or on lookup on the two nearest neighbors?) that might make this overall a win.

Making clauses one type of instruction may seem weird and limiting, but when you think about it, a lot of code looks like:

- execute multiple loads
- compute
- execute a few stores

We tend to interleave these on CPUs to maximize IPC, but the interleaving is not logically necessary, especially if you have enough registers that you can perform all your loads up front, you don’t need to worry about spilling to stack and similar such issues.

Even things like load clauses work well for many purposes. You usually have no internal dependencies within the load clause, so you can simply fire off one set of loads after another after another, then delay launching compute until all the loads have moved from cache/DRAM into registers.

An instruction in L1 is accessed by address (ie PC) as expected, but once the first instruction of a clause

is accessed, the instructions of the clause are placed in the L0 caches of the appropriate unit. In those L0 caches they can be referred to by a short number (let's say 8 bits long) rather than by address. This means that flow control (eg jumping to a new function, or looping back to the start of a loop) consists only of the Instruction Sequencer that's building clauses and handling flow control telling the instruction loop that the next thing to do, after clause X is done, is to start execution of clause Y [identified by this few bits identifier].

What all this ultimately means is that moving instructions around (ie I-fetch) uses almost no energy.

- instructions are shared across 32 lanes
- pre/decode does some work which is cached in L1
- what we might think of as Fetch consists of checking that the instructions of a clause are in L1 and moving these instructions to the appropriate L0 in front of each relevant unit

Clause based execution has been used by AMD for a long time, and they describe it a little (though not much) in the RDNA white paper: <https://www.amd.com/system/files/documents/rdna-whitepaper.pdf>
 Elements in common with Apple include the separation of control flow from “data” instructions, and the uses of clauses of a particular type of instruction. But AMD (as far as I can tell) have not pushed the idea as aggressively as Apple in terms of how much instruction execution is decentralized

instruction flow in detail

Instruction flow seems very strange until you think about how a GPU differs from a CPU.

Obviously we want instructions to be cached, for the usual reasons, and obviously instructions are repeatedly reused (think for example of a particular shader that will be applied repeatedly to thousands, even millions, of pixels, which, even when considering 32-wide SIMD, means constant re-execution of these same instructions).

So we want an L2 cache (for the entire GPU, for size and to communicate between GPU cores).

And we want a standard L1 I cache for each core, with the usual cache lookup stages to handle virtual translation and jumps (eg function calls or loops) and multiple cache ways. Fortunately all instruction lookup is amortized across 32 SIMD lanes.

BUT what happens next, after the L1I?

One option is to run the L1I cache the same way as a CPU, and to deliver the instructions to each lane each cycle. This is obvious, but it means that we get no instruction re-use in terms of energy. Each cycle we pay the cost of loading the instruction from L1I, “duplicating it” across 32 lanes, and sending it to the relevant execution unit.

Can we reduce some of this overhead by using concepts like a loop buffer, a trace cache, or a micro-code cache? These are all ways to try to reuse or simplify cached instructions.

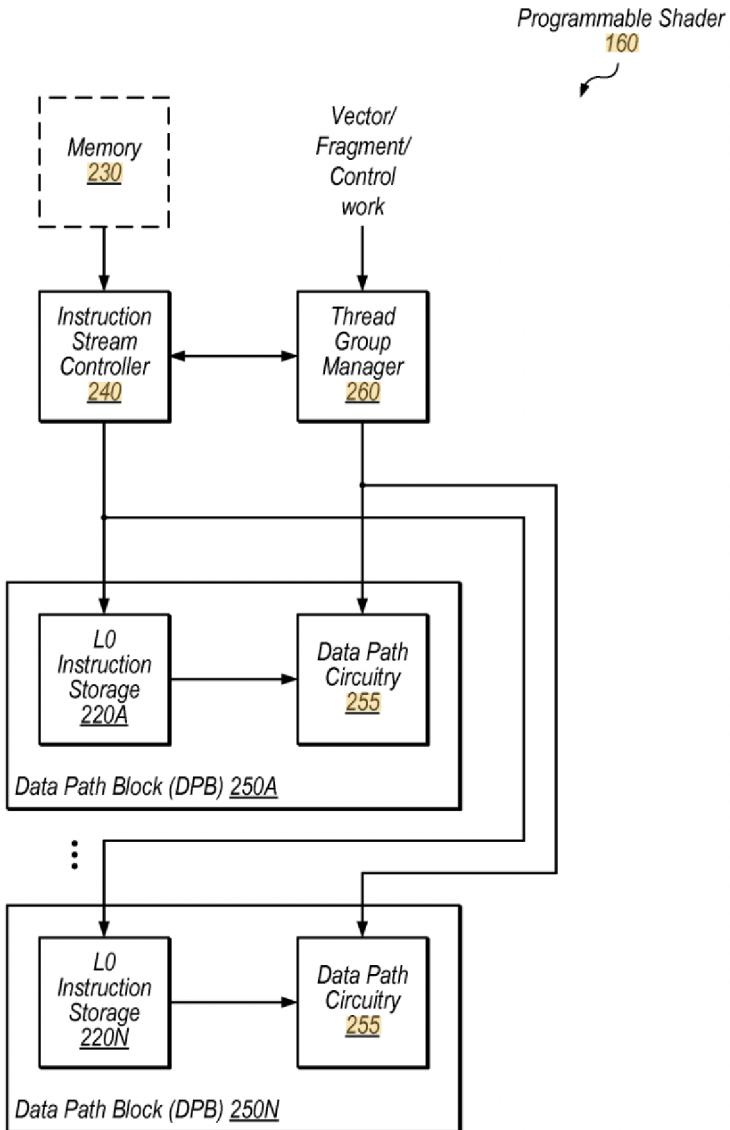
clauses

Apple’s version of this I-caching is the use of *clauses*. The first Apple patent discussing this appears to be (2015) <https://patents.google.com/patent/US20160371810A1> *GPU Instruction Storage*; although

clauses were already in use by AMD and, I don't know, may have a long history of use on early GPUs and accelerators.

A clause is essentially something like a trace, a short, four to sixteen or so, run of instructions with no control flow (predication is OK).

The L0 cache holds a few (initially four to sixteen or so, now I would guess more like 64 or so) clauses. Hence the instruction flow is



- The “Thread Group Manager” (TGM) maintains a pool of active warps (6 or so per quadrant, which are actively being switched between) and pending warps (20 or so per quadrant, which have register state

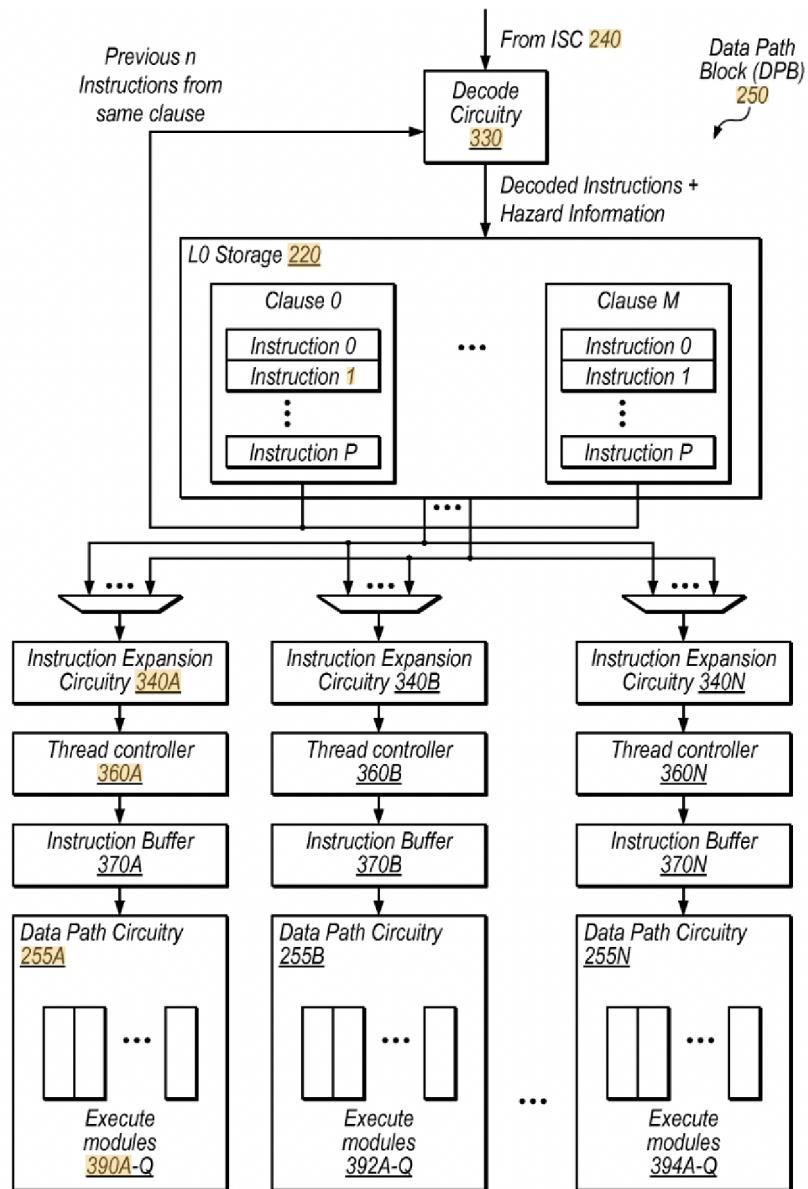
available, but which are or were waiting on memory, and which will be switched in when one of the active warps has to wait on memory). So this TGM may decide that a new instruction flow needs to be pulled in. If necessary requests go out to L1I, L2, or even DRAM.

- instructions are pre-decoded from L2 to L1I
- they flow from L1I through an “Instruction Stream Controller” (ISC) which, dynamically, groups them into clauses.

I think that, more or less, TGM decides about overall flows of instructions (ie load instructions for this warp of this threadblock of this kernel, now switch to instructions for that warp of that threadblock of that kernel), while ISC handles the internal details given a single stream of instructions. Also I think the sharing is a single TGM and ISC per core, but the part called a DPB is per lane (so replicated 32x per quadrant). This works out fairly well in that the work done by the ISC and TGM only needs to happen occasionally, for many cycles these units can sleep, because most of the time the four shader quadrants will be executing straight line code from the L0I clause caches.

As we will eventually see, one of the many changes in the A17/M3 GPU is that it becomes superscalar, capable of executing, within a single quadrant, possibly two instructions per cycle, from each of two separate warps (ie independent, so not need to worry about hazards between the two instructions). This presumably has some impact of the L0I clause caches (maybe they become 4-way banked to allow two-way instruction access per cycle?) and perhaps also on the ISC (now two per core, to handle the increased instruction flow?)

- the ISC processes the instruction stream which, apart from grouping the clauses and handling control flow
 - + decodes instructions in the clause
 - + detects and marks hazards within the clause (ie if one instruction in the clause depends on the output of an earlier instruction in the clause, so can't execute until that earlier result is available)
- clauses are “registered” with the “Thread Group Manager” which then handles thread scheduling by dispatching the entire clause+warpid to the lanes. At this point the clause consists of decoded instructions together with marks about dependencies, which will mean things like “instruction 3 cannot execute until at least 8 cycles after instruction 1”
- each clause is sent to the L0 of the appropriate “execution unit” of the target quadrant. The diagram does not make it clear, but the idea seems to be that different “execution units” (eg load store vs FP arithmetic vs texture) of a quadrant all have their own L0I clause caches, and so the N index in the diagram above loops over these units (m units in each of four quadrants).



- conceptually instructions flow from the L0I into a specific “execution unit” and then into lanes; In the diagram above we are seeing the L0I and execution units for something like “floating point execution”. So the execution units A..N are something like “FP32 execution”, “FP16 execution” and “integer execution” engines, each 32 lanes A..Q.

If this were the “load/store” execution unit, maybe there would be a “load execution” engine and a “store execution” engine, each with maybe 8 lanes wide.

- however some complex instructions are cracked or microsequenced at the head of each execution unit into a few microcode instructions. The patent explains why having this cracking done after, rather than before, the L0 is actually optimal, all things considered. It costs a little more energy, (but the instructions are rare) and it saves area.

The Thread Controllers handle the sequencing of instructions through the successive instructions of a clause. This is not completely trivial because some instructions may have been expanded in the earlier Instruction Expansion Circuitry stage. Then, when a clause is complete, TGM needs to be informed so that sequencing of the next clause can begin

Not completely obvious in this is that flow control is handled *between clauses*.

While predication handles simple per-lane “if-then” behavior, more complex flow control like a function call is handled by the front-end without the shader core doing anything; what the shader core sees, in a sense is a stream of “super-instructions” in the form of clauses, and each lane executes the super-instructions from the lane’s local clause cache one instruction of the clause at a time.

Probably the most common form of flow control is looping where, at the end of the clause execution, the shader will be told to execute the same clause again.

It is probably also the case (though I’ve never seen it stated) that even once preemption was added to the design, preemption points (ie interruption points) are between clauses, rather than between instructions.

Recall that nVidia handles hazards by compiling, into the instruction stream, any delays that might be required to handle hazards (ie something like “instruction 3 cannot execute until at least 8 cycles after instruction 1”. That’s obviously the minimal overhead way to do it, but Apple’s scheme is also fairly minimal overhead (once for the clause, then amortized over possibly many reuses of the clause). nVidia’s scheme also requires separate handling of variable length latencies (ie load instructions).

The Apple scheme seems to be

- + to provide fine-grained dependencies within a clause (this hazard marking already described)
- and then
- + coarse tracking of dependencies between clauses.

So rather than worrying about fine details of “instruction 3 of clause A depends on instruction 2 of clause B” we simply mark something like “clause A depends on clause B” and wait until every aspect of clause B is complete (all instructions, possibly any loads, etc). This may delay clause A longer than

strictly necessary, but is good enough (hopefully there are other clauses from other warps to run while we wait) and requires much less overhead.

We don't know the precise details of how this works in all cases, but for the most important case of Load, we do know that there is a Load_Wait instruction which is presumably placed by a compiler as the very last element of a load clause. If we ensure that similar instructions are placed as necessary at the end of other clauses (perhaps this might be necessary, eg, for vertex or texture or ray clauses?) then that might be good enough – the next clause cannot start until the previous clause is complete, and complete means that however data might be communicated from one clause to the next, via register or via memory, has progressed to a stage that the next clause will see the data correctly.

This scheme manages to give us most of what we want.

- most instruction sequencing is cheap; we just move to the next instruction in the clause without any more complicated cache lookup.
- the (decoded+hazard-marked) instructions in the L0 clause cache can be (and are) reused repeatedly simply by having the Thread Group Manager pass a short (few bits) ID the next time we reuse the clause. We expect frequent clause reuse not just because of loops but because of multiple warps within a threadblock, and multiple threadblocks within multiple batches of a kernel grid.

Clauses seem to serve three main roles.

- Act as “atomic” super-instructions that do multiple things over multiple cycles, but don’t cost any sequencing, hazard detection, or decoding resources.

The hazard detection, especially the way dependencies on loads are handled, would suggest (as do many subsequent patents, though it’s never fully stated) that clauses hold only one type of instruction (eg texture, load, store, or ALU), and presumably the compiler tries, so far as possible, to aggregate long runs of instructions of the same type.

Once this is accepted, then it makes sense to have, in fact, multiple L0 clause caches, one for each of the different execution unit types, and we will see that this is the case. So I kinda handwaved and spoke of a per-lane L0I cache, but assume that, for example (like other GPUs) there are 8 load/store units, and each one executes an instruction four times for four lanes. Then there would be 8 “per-lane” L0I caches holding load/store clauses.

Meanwhile higher-level sequencing by the Thread Group Manager consists mainly of, every few instructions, telling all the lanes, OK, now switch to execution unit X and start executing clause Y, based on some combination of scheduling algorithm (fairness, oldest thread first, whatever) and when clause Y has become runnable because the earlier clauses on which it depended have completed. Once we start execution of the clause, the unit (load/store, FMA units, whatever) runs for a few cycles autonomously, before reaching the end of the clause and needing to be given a new execution unit + clause ID.

Clauses are not atomic in the sense that they take a single cycle. But they are atomic in the sense that they form units of tracking. Dependencies exist between clauses, break/switchpoints occur between clauses. If a clause begins execution, it will finish execution before something else (like some sort of preemption) happens.

- Act as cacheable units which can be reused, either within a loop body, or via successive warps of successive threadblocks of a kernel.

Thus the L0 caches acts like a combination of trace cache and loop buffer.

The claim in various places on the internet seems to be that the M1 GPU's L1I is 12KB which seems smaller than the competition (generally 32KB), but maybe the ability to reuse clauses across thread-groups means this is not as bad as it seems, given that so much GPU work consists of data-independent threads but executing the same code? Maybe there's usually work to do from within the L0I's while data is being reloaded from L2 in this (smaller than in other GPUs) L1I?

- Instruction sequencing (ie control flow, and fetching the next instructions to feed into the Instruction Stream Controller; probably also preemption) is handled outside the quadrants, and can be shared across the quadrants.

Another angle on this is remember that, at any given time, a quadrant maintains two different sets of "executable" SIMDs.

The first is the *active* threads, perhaps about six or so, any of which at any time (more or less) can be swapped in if the currently executing thread is stalled for whatever reason. These threads have some degree of state associated with each of them and are essentially tracked by clause.

The second is a rather larger (up to perhaps 24 or so) set of *potential* threads with slightly less associated state (though their registers are live), and tracked by PC rather than by clause. Part of their becoming active (eg if one of the active threads blocks on memory) is using the PC to load instructions into the L1, and then proceed to clause-based execution.

Note that, among other simplifications, this all means that only the second thread scheduler, the one tracking PCs, needs to worry about I-cache misses. By the time a thread becomes active, the clause it's going to execute has already been fully loaded all the way from DRAM through L2 and L1 to L0, so no I cache miss can occur once it starts execution. Remember as far as energy goes, it's all about trying to simplify the most repeated path (the stuff that replicates across 32 lanes and executes every cycle...).

At this point you may want to read <https://chipsandcheese.com/2023/12/04/gcn-amds-gpu-architecture-modernization/> which discusses AMD's GCN (the design before the current CDNA design). With the Apple design in mind, you'll see how elements of AMD's design match Apple's design. I think many elements of the GCN discussion have been simplified to the point where it's hard to make sense of them, and the clause element of the design is not mentioned. I think if you interpret the design as a variant of how I described Apple's design (split instruction stream into execution-unit-specific clauses, store these clauses in buffers, each execution unit looks in its clause buffers to find if the next instruction in one of the clauses is executable) the design makes overall sense. The set of two buffers per five execution units at an abstract level looks like a pool of ten warps from which we can schedule any runnable instructions, but I think seeing it at the lower level of two buffers per clause makes the strengths (and weaknesses) of the system more clear.

The differences from Apple include the specific details of the execution units available (GCN has vector and scalar datapath, Apple appears not to have scalar; Apple appears to split memory opera-

tions into separate load and store clauses); and that Apple retains clauses for reuse in an L0 cache, whereas GCN (at least if the article is correct in this respect) temporarily holds onto a clause in one of two clause buffers per execution unit, but the clause is replaced with another clause as soon as its execution is complete.

You can see the AMD evolution from a VLIW instruction (ideally needs to specify five different instructions for the five different execution units in a single bundle, with complications in terms of getting the timing and dependencies just right.

The GCN scheme allows for the same end result (up to five execution units all issuing an instruction in a cycle) but with more timing flexibility since each clause comes from a different warp and proceeds on its own schedule based on its particular dependencies.

The Apple scheme has those same performance advantages along with

- a choice of more clauses from which to try to find a runnable instruction each cycle (L0 cache rather than two clause buffers)
- power reduction (clauses get reused from the clause cache rather than repeated movement from L1 through decoder to buffer).

You'll see that GCN also splits its core into smaller units (the sort of thing I've been calling a quadrant) and it's likewise unclear exactly how independent quadrants are. The article states that they are essentially as independent as nVidia's quadrants, so separate register files and dedicated routing of specific warps to specific quadrants.

This is clearly a design decision with many aspects to recommend it, and it's been my assumption that Apple (at least as of M1 and M2) operates that way; but it's not the only choice.

You can instead operate a unified (4x larger) register pool, a set of unified (4x as many entries) L0 I caches, and each cycle each of the 4 SIMDs tries to find an executable instruction from one of the set of clauses stored in the unified L0I clause cache. Or you can adopt a viewpoint somewhere between these two, like a unified register pool and clause caches, but warps statically bound to a particular SIMD and its per-SIMD operand caches.

There's also a whole lot of messiness in terms of per-lane instructions that are vector instructions (so a lane is like a 4-wide NEON instruction; and fitting wide warps into many fewer lanes that execute over multiple cycles). I'm ignoring all those elements because nVidia very sensibly got rid of the worst of them, and then everyone else followed, so they have no relevance to Apple's design.

(Before you criticize GCN too much, remember it's from 2012! I expect that many of these limitations have been resolved by now in CDNA, to be more like Apple.)

After seeing even more details below of how Apple does things, you may want to look at the AMD scheme a second time.

clause chaining

(2016) <https://patents.google.com/patent/US20180067748A1> *Clause Chaining for Clause-Based Instruction Execution*

Recall the 2013 patent described how we have 4 active threads at any one time, and that they

execute in a lockstep rotation (as a barrel processor). Effectively each pipeline stage has four slots, and each independent thread lives in one of those slots. Also in the 2013 design, the operand cache (and a few other state elements like the predicate mask) are associated with these slots in the lockstep rotation of the barrel processor.

Now put this together with the clause design described above. Every time we switch out one clause and switch in another, in principle the clauses could be from different threads meaning we

- have to flush the relevant elements of the operand cache (which are tagged by the slot, not by something more abstract like a threadID) and
- we have to delay long enough to move the previous predicate mask to longer term storage, and swap in the predicate mask for the new thread.

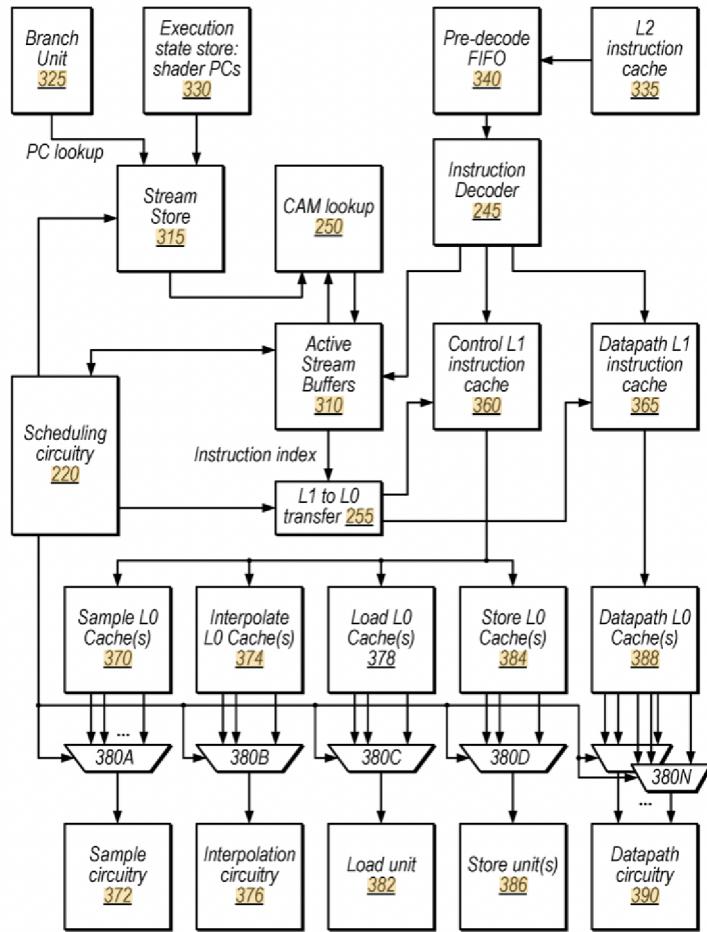
But this is clearly sub-optimal! If we can schedule things (and the Thread Manager will try to do so) to follow a clause with the next clause from the same thread, then both the operand cache and the predicate mask will remain valid for this slot and can be reused for this next clause. This *clause chaining* is what the patent is about.

(Obviously many of these details are no longer relevant as we move to the more modern, more flexible design which is no longer barrel based. But it's always interesting to see design evolution!)

scheduling based on clause type

(2017) <https://patents.google.com/patent/US10324726B1> *Providing instruction characteristics to graphics scheduling circuitry based on decoded instructions* is the next step in this series. The diagram below looks complex, but you can see that it confirms elements we have already seen, while adding some other pieces.

Confirming what we already know, we see that instructions flow from L2 through pre-decode and L1 to multiple L0I clause caches. Sample and Interpolate are elements of texturing, and Datapath means what we'd think of as arithmetic, so FMUL, FADD and friends.



Let's look at this in some detail.

The Pre-Decode FIFO aggregates instructions into linear streams. Recall that at any given time instructions may have been requested (or even, perhaps, prefetched) for multiple kernels, and instructions can arrive out of order from L2 or other memories. So the FIFO sorts them by instruction stream and waits for the stream to be fully populated in-order with no holes.

The Instruction Decoder is a full decoder (including checking for hazards within a clause), and has moved from after L1I to before L1I. This means we now have a large store for fully-decoded and hazard-checked clauses, limiting how often these have to be reconstructed. This also reduces latency once we

need to move a clause to L0, since only movement is required, with no cycles for the additional Decode+Hazard check.

The Instruction Decoder also (like a traditional Pre-Decoder) classifies and marks instructions into instruction classes.

The L1I cache structure looks a bit weird, but makes sense when you think about. Control flow and looping are handled outside the shader core as scalar, not SIMD instructions, and are infrequent so a common “sequencing” unit can handle four quadrants.

Sequencing (things like the Branch Unit, the Execution State Store [which records per stream PC] and the Stream Store [information about sequences of clauses per stream]) needs to look at the instruction stream to decide how to modify the PC, but it does not need to look at any instructions apart from the control flow instruction. Similarly building and transporting clauses to L0I does not need to know about Control Flow instructions.

So we can save a little power if, rather than the obvious I-cache design (where reading a control flow instruction will activate various other uninteresting bytes) we design the cache more as something like a set of straight-line traces (the Datapath L1I) and a “control index” into those traces which is essentially the Control L1I.

Sequencing will involve

- looking at the next element in the Control L1I, calculating the new PC resulting from the control flow instruction,
- looking up the PC in Stream Store (which handles starting new streams or jumping to a new stream [as when we construct Indirect Command Buffers]). At some point these new streams will be loaded in and activated, but there is some flexibility in the timing, with new streams not loaded immediately, if the core currently seems active enough. These “virtual” pending streams are stored in this Stream Store.
- if this isn’t a new stream, we lookup the PC in CAM lookup, connected to Active Stream Buffers, which essentially knows what’s in the L1 Datapath cache and in the various L0’s
- if there’s a pre-existing clause, we send that clause ID to be executed next as appropriate, or
- we transfer the clause from L1 to L0, or
- we send a request for new instructions to L2.

There’s an implication that the separate Control I cache can also be used as the basis for a very simple (but reliable) prefetcher. We can scale the Control I cache to be somewhat larger than the Datapath cache so that it holds a larger footprint than the Datapath cache.

Imagine a slightly more sophisticated, pipelined, version of the PC calculation which, instead of

- Calculate and Handle next PC
- Lookup and transfer next clause,

we switch to

- Handle PC for next clause
- Lookup and transfer clause A
- Calculate next PC for what will happen after clause A

If we do this and if the instruction for this next PC is available in the Control I cache, then under most conditions we can lookup if the PC for clause A+1 is available in the Datapath I cache, and if not, send

out an L2 request for the instructions, to load while clause A is executing. This works because we expect the elements of the Control I cache to be small and infrequent, so we can give the cache a much larger footprint than is feasible for the Datapath I cache.

The Scheduling Circuitry makes occasional decisions about which, from the pool of clauses available in L1I, to move to one of the L0s. As opportunity arises (most obviously when a clause completes) we want to move a new clause into the shader core. We have a pool of clauses available based on the currently active or pending warps, and which clauses have had their dependencies resolved (eg finished waiting for memory). The usual sorts of options present themselves, eg round robin, or choosing the oldest warp.

However Apple introduces one more element into the decision, namely the type of unit on which the clause executes.

Suppose we have two possible clauses ready to run, and the opportunity to schedule a new clause arises. One of the clauses is a load clause, one is a datapath clause.

We can look at the level of activity in the various execution units (each of which is, each cycle, making the lowest level scheduling decision as to which instruction to execute next from a pool of say three clauses from three different warps). If we see that the datapath unit is executing at about 95% activity, but load is executing at about 15% activity, then it makes sense to schedule the load clause rather than the datapath clause.

The important point is that we see high level scheduling based mainly on *the ordering and QoS* properties of a kernel and when cores become free (to try to maximize data reuse on a core or chiplet).

At the other end, within an execution unit (eg FP) we schedule the next instruction based on which instructions (from a few active warps, active meaning have clauses in the L0) are *runnable*.

In between these two levels, at the level of the clause, we try to ensure (as much possible, which will depend to some extent on having many different kernels active) that *the different hardware units* of a quadrant all have a few clauses lined up so that they are all able to execute instructions simultaneously.

Note that this all suggests that the question “how many instructions per cycle” or “how wide” is a quadrant is not the full story. Instruction issue is very distributed, so that it would seem that, for example, if you created a workload with enough simultaneous kernels doing different things, you could have clauses for texturing, for load, for store, and for datapath all executing in the same cycle.

For nVidia it’s fairly clear, given the design, that every cycle we issue one instruction; that instruction takes a minimum of two cycles (16 FP lanes or 8 load/store, but 32-wide warp) to “internally” issue across the entire warp, with the net result that you frequently can get what looks like two or even three simultaneous instructions.

For Apple it’s more like, to use very crude numbers, every four cycles we issue a four element super-instruction to an execution unit. With four/four this would, on average, give one instruction per cycle. But suppose now every four instructions we issue an eight element super-instruction. Now we have the potential to, on average, be executing two instructions (say a load and an FP) every cycle.

The exact IPC we will see depends, obviously, on the mix of clause types and lengths; but also on

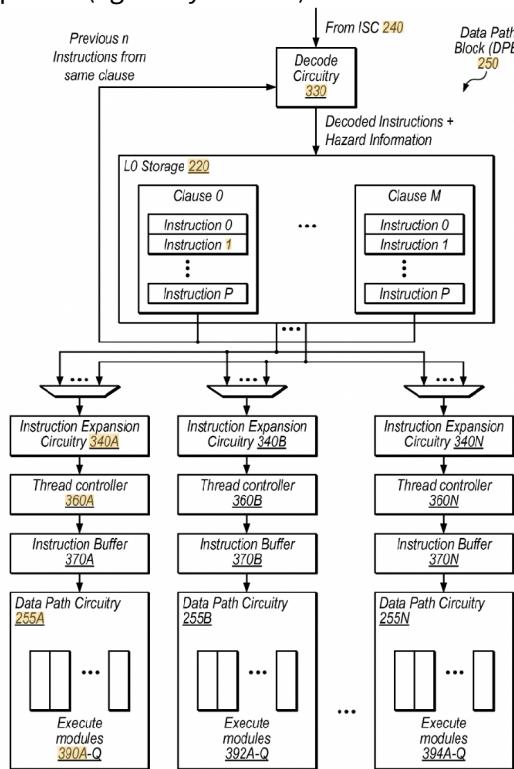
details like whether control-flow calculation ever becomes a bottleneck, or whether this control machinery (how often we need to transfer new clauses to an execution unit, how many clause-completions we handle per cycle, etc) is fast enough to scale to four quadrants without throttling.

The primary new points are

- decoder before L1, not between L1 and L0
- as instructions are (pre)decoded we add some flags that describe the type of unit in which the instructions execute
- this information is used by the Scheduling Circuitry as part of its input (along with other factors like number of operands and operand cache state), essentially to avoid overwhelming limited throughput units.

Another thing worth thinking about at this stage is exactly how the quadrants of a core operate. My assumption has been that quadrants are largely independent, a set of registers, L01 caches and execution units, that operate autonomously, much like nVidia split a core into (at different times) halves, sextants, or quadrants.

But a different way of viewing things is that suppose each of these L0I's is “large enough” and multiported (eg 8-way banked). Then we could interpret a diagram like



as a single core-wide L0I (for eg datapath, ie FP) instructions capable of feeding four independent FP pipelines by reading four instructions per cycle from four different clauses. In this model, we have a single unified pool of core-wide registers, a single unified pool of L0I's, and the illusion of quadrants primarily reflects the fact that some execution unit types (certainly datapath, maybe also load store,

but perhaps not some of the graphics specific hardware like vertex processing or texture processing) are four-way replicated?

Certainly this is the model one would hope holds if cycle time allows for it, because it's capable of better load balancing than four independent quadrants.

scheduling based on finer details.

Once you have the above idea of scheduling clauses based on how busy each execution unit is, where else can you use it?

Consider stores. Stores can be placed in a variety of different locations, for example local storage, global storage (ie the L1D cache), texture storage, vertex storage, etc. In devices at this time (around 2017) some of these storage locations have lower bandwidths than other locations. We this have the same issue of balancing bandwidth – we would prefer to balance clauses writing to fast storage along with clauses writing to slow storage, rather than scheduling multiple simultaneous clauses writing to slow storage.

This is covered in (2017) <https://patents.google.com/patent/US10452401B2> *Hints for shared store pipeline and multi-rate targets*. The details are slightly different, but the basic idea is that we

- track how full the storage queues are to various storage targets, and
- based on statistics we can predict the storage target of a given clause
- so once any particular storage queues get full enough, we try to choose instructions from clauses that are predicted not to target the full queues.

Done optimally, this scheme would also extend one level higher, to clause scheduling, to try to ensure a balance of store clauses that target different address types. This would certainly fit into the model of the previous patent, but it's not explicitly called out.

context switching and preemption

early virtualization (2012)

On an apparently separate trajectory, we start with (2012) <https://patents.google.com/patent/US9727385B2> *Graphical processing unit (GPU) implementing a plurality of virtual GPUs*. The use of the term *virtual* here is confusing, but makes sense in light of the history I gave of GPUs generally, and nVidia GPUs in particular.

Essentially what this patent wants the GPU to provide is the full package to allow *different* processes to use the GPU *simultaneously* without getting in each others way (something nVidia inched towards over the next decade). Aspects of this include

- being able to run kernels from different processes more or less simultaneously
- support for simultaneous separate address spaces for different client programs (ie some sort of MMU and TLB support)
- some sort of preemption control (so that, if necessary, a background process can [temporarily] be forced off some cores)

- some degree of QoS

All this is obvious from the CPU side, but somewhat novel on the GPU side.

Consider the per-process address space issue. For a CPU (think of an SMT CPU) we handle this by associating a VSID (virtual space ID of some sort) with each thread. Each thread of the SMT CPU presents its VSID+ virtual address to the TLB, and TLB lookup operates based on both of these. Then the data cache operates in physical address space. This scheme allows two (or more) SMT threads to run together, and something similar could be used for the GPU. Note that, at this time of 2012, the MMU appears to be envisioned primarily as an address mapper. Specifically, it does not seem to be set up to handle *demand-fault paging*, with all that implies (detect a page fault, send an interrupt to the OS, suspend state while the fault is serviced, restart the instruction); instead the expectation is that GPU pages will be wired down, at least for as long as a particular kernel makes use of them.

The QoS issue has to do with not wanting say a large GPU Compute task to lock up the GPU for long enough that what needs to be drawn on the screen lands up being delayed.

Beyond the fact of this aspiration (of course now fully realized) the details of this particular patent are less important.

The solution at the time, as I understand it (and I am probably missing many details), is to give the GPU many “contexts”. The patent suggests initially three or four, but I believe modern Apple designs now have 16, maybe more; one of which is locked to the OS, while the others are for general use.

A context is essentially a VSID, which distinguishes between different address spaces and allows TLB entries and suchlike to be shared by simultaneously executing code; along with some other per-context registers including, eg, QoS register(s). In the patent a context is also the unit of QoS, ie a context (ie a specific process) has a higher or lower priority relative to what else is going on in the GPU. This may seem obvious, but it may not be optimal (what if, within my particular process, I want to give some kernels high priority and others low priority, the way I can give my threads on the CPU separate priorities not specifically tied to a single process priority).

You would think that by now this has been sorted out, but I've seen no discussion of the issue either way.

Another way to look at this is that, in a default dumb GPU, every piece of storage and state is implicitly associated with the one and only controller process. Virtualization attaches appropriate context tags to every piece of state and storage so that the device can be controlled on a per-context basis – we can do things like “pause all kernels and threadblocks with context n ” or “invalidate all registers and Scratchpad storage with context n ” or “be aware that these resources are reserved, allocated to context n , and not for use by any other context”.

The patent also suggests that, at this early stage, apart from the issue of preemption, QoS is primarily handled by scheduling at the threadblock level, that is higher QoS kernels preferentially get more of their threadblocks handed out to GPU relative to lower QoS kernels. Compare this to more sophisticated QoS schemes (which are mentioned in passing, but I suspect at this point aspirational) which

also incorporate deadline information, so that kernels which must complete by a certain time get boosted enough to temporarily take over the entire GPU.

Another scheduling feature suggested even at this early stage is scheduler awareness of both specialized hardware in a core, and kernels which exploit that specialized hardware, so that if eg texturing is currently going unused, a lower-priority texture threadblock might be scheduled anyway, just to take advantage of the free hardware. Like deadline scheduling, I suspect this is, at this early stage aspirational rather than real.

co-operative context switching (2017)

We have seen with GPUs essentially a recapitulation of what happened with CPUs/OSs.

In the PC world, the first OSs (think the Apple II or DOS or even the very first few versions of MacOS) a program ran to completion before another program could begin.

Next a combination of conventions in the OS and in a program meant that a program could context-switch at particular well-defined points in the program. This is the world of Windows 95 or Mac System 7. It's adequate for some purposes, but things can go wrong if the program is poorly written (or has bugs) and so refuses to provide a yield point at least every second or so.

Finally we get genuine context switching whereby the OS can grab control from any program, regardless of what it is doing, any time this is required.

In the GPU world the equivalents are that we begin with the GPU as a resources owned by one program (generally a game) at a time, and no sharing, no use by another program until the first program quits.

After that we have various models that essentially operate by having the OS request the GPU yield, and the GPU responds by no longer submitting new threadgroups and waiting for the current threadgroups to complete.

This has the potential to cause problems with long-running shaders, so the idea has been to have the compiler insert potential syncpoints (the equivalent of something like System 7's `WaitNextEvent()`) in code that looks like it could be long running. At these syncpoints, the state of the machine (some sort of checkpoint) would be stored, so that later we could restart at these checkpoints.

This might seem like a clever solution (apart from time wasted on instructions that don't advance the shader computation), but there's an unexpected GPU-specific issue: what to do about lanes that are not active? You may have, for example, some predicated lanes via a function higher up calling a potentially long-running function. Functions in the non-active (non-predicated) lanes will not execute the syncpoint instruction, unless we engage in some additional hacks to force those instructions to be treated differently.

Or should we just accept that sometimes there will be a long delay before a context switch, and hope it doesn't happen too often, just like Windows 95 and System 7.

The above appears to be the way things were done in the first few Apple GPUs. For example the already-

discussed patent (2018) *Distributed compute work parser circuitry using communications fabric*, in its context switching discussion, assumes context switching based on preventing new workgroups from being scheduled, and waiting for existing workgroups to complete, with no way to *force* interrupt them.

Even so, we can try to make priorities work better, by treating the issue (at an abstract level) as a priority inversion problem. ie raise the priority of the blocker! In the case of the GPU, this can be done by boosting the frequency of the GPU, and also, if appropriate, the speed of the NoC and DRAM, to reduce the time till the lower-priority GPU code hits a sync point or exits.

This is described in (2017) <https://patents.google.com/patent/US20190057484A1> *Fast GPU Context Switch*.

fine-grained context switching (2020)

But of course we'd prefer to have modern style, low-latency context switching. We eventually get this in (2020) <https://patents.google.com/patent/US20210224072A1> *Instruction-level Context Switch in SIMD Processor* where we get a mechanism that's essentially like a CPU, namely the ability to interrupt a kernel at some point, wait for all the in-progress instructions (most obviously loads that missed in cache) to complete, then execute a handler to store state.

This preemption/context switching mechanism builds on elements we have already seen. Here's my guess as to how it works, based on the clause design and the patent.

A request for a preemption starts somewhere higher up in the system and flows down to the high-level branch sequencer so that, instead of the next clause being sent to the store unit, a special clause is created with properties of, among other things, that it depends on all previous clauses. Now essentially the machine will drain while all the other clauses complete and this clause can't execute until they are complete (and so all prior operations, including loads, have completed, and register state is stable).

The new store clause will consist of one or two or three special instructions that get expanded by the in-line expansion mechanism into instructions like "store these general registers at this address in global address space" and "store these special purpose registers in global address space". Once the clause is complete, execution of this particular warp has been suspended.

This saves the state of one warp. We will have to do this for all warps we plan to suspend. Thread-blocks themselves have associated state (which warps are in process, which are completed) and perhaps some Scratchpad storage, and this may also have to be saved. This is done at a higher level, in the circuitry that dispatches warps from any particular threadblock executing on this core.

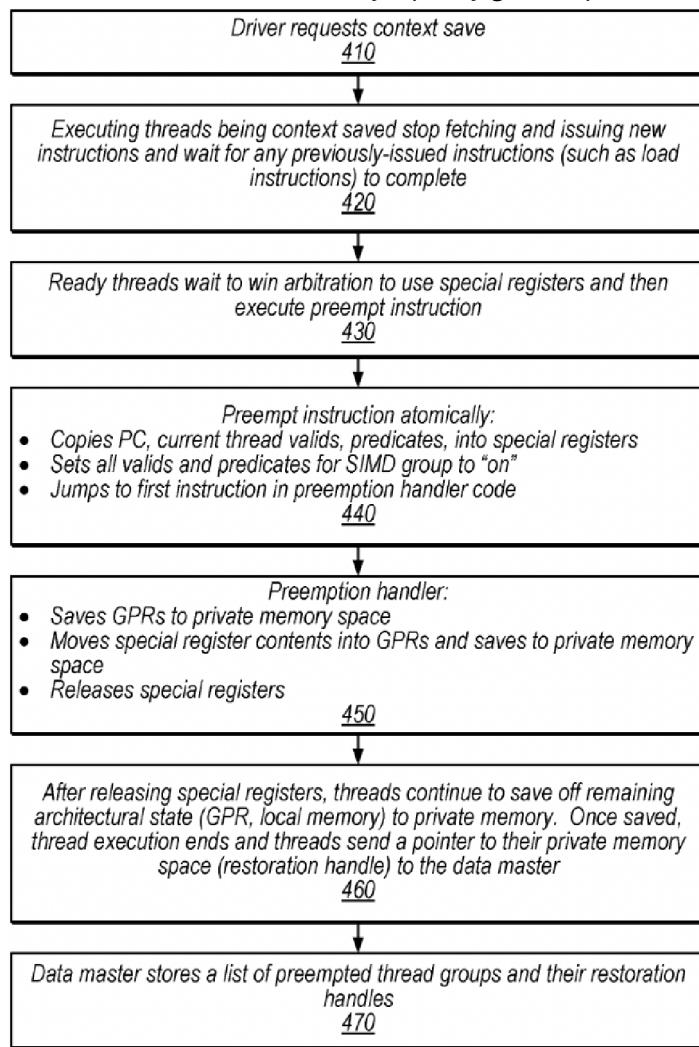
Restarting a suspended warp happens in the same way in reverse; first we restore threadblock state, then for each warp to be restored a clause is constructed, now with special purpose load instructions that load all this previously saved state.

In a sense this is all obvious – preemption and context switching have to involve storing and loading state. The part that's interesting is that, if we use the mechanisms already in place, we can do it fairly easily without disrupting the existing design, without disrupting other simultaneously executing warps that we don't want to suspect, and in parallel across all warps we do wish to suspend, each suspending

at their own optimal pace without central co-ordination.

There is one final technical tweak that's needed to this scheme, which is dealing with the already mentioned predicate mask problem! If you just start executing context-save code, only the predicate-active lanes will save registers!

The flow chart below is actually a pretty good explanation:



The two technical tweaks are

- the very first instruction of the newly constructed “preemption clause” is a preempt instruction which accesses various special registers. These special registers (which include predicate mask) are saved to the context storage space, and then the predicate mask is cleared. The reverse is done on predicate load.
- the second tweak is that there’s rarely any need for the load or store instruction to access these special registers, so we don’t provide high performance parallel access to them. Instead there’s an arbitration process whereby one warp at a time gets access. This doesn’t hurt performance much because there are not many of these registers, and as soon as they have been stored, the mutex to access special registers is released.

I’ve suggested that the millicode that performs preemption handling (reading +handling special registers) and context switching (saving registers to memory) is triggered by special functions in the newly created clause.

Another way to do this, suggested by the patent, is to embed these two functions (preempt and contextsave) [in fact four functions, each of the two in a load and store version] in each kernel binary; then instead of creating a synthetic clause at preemption time, the high-level instructions sequencer simply sends down these two clauses to the store/load units. This scheme makes each kernel binary a little larger, but you can tweak these context save/load functions to only save/load the actually used registers.

You can imagine alternative ways of doing this even more efficiently. For example you could define a register save/restore mask that gets set at appropriate points in the code (maybe on function entry/exit). [In fact Altivec did something like this back in the day using the VRSAVE register.] This mask could then be used by either the per-binary contextsave/load code, or we could now hard wire that code into millicode and make each binary a little smaller.

Remember that even though context switching is possible, it will still be expensive! My guess is that the GPU has heuristics to try to wait for warps (and if possible full threadblocks) to complete and drain. If a warp is complete you don’t have to save its state (ie registers); if a threadblock is complete you don’t have to save its state (which warps have completed, maybe also the Scratchpad storage).

We only forcing a context switch if the threadblock does not quit/yield within a particular duration.

Instructions

Let’s take a small digression to look at the details of a few instructions which provide unusual functionality, or perform their tasks in an unusual way. Some of these give general insight into the current design; some are just strange and interesting!

speculative GPU predication

This first discussion seems pointless and obsolete. But ignore the details, the interesting part is the idea of how latency reduction is implemented.

(2013) <https://patents.google.com/patent/US9633409B2> *GPU predication* gives a few technical details about how predication was implemented (at least back then, maybe still today). Consider the handling of back to back instructions, and suppose that a particular instruction has a 6 cycles latency (not unrealistic). Then if the next instruction uses the result from that earlier instruction, we have to block for at least six cycles.

Obviously with any in-order CPU, the first line of defense is to try to get the compiler to schedule dependent instructions interleaved with other independent instructions, but that's not always possible. With a GPU the next line of defense is to schedule in one of the other warps during these dependency cycles. However the patent hints that, at least at the time, the GPU only supported three simultaneous threads; so there was limited independent work available on that particular front. Which means that there's value in trying to work around this back-to-back dependency constraint.

What Apple does in the patent is to make execution “speculative” when necessary.

An instruction will know that it is dependent on a predicate, and that the predicate has not yet been calculated. Under strict rules, that might mean the instruction should not begin until the predicate mask is known. But do we have to be strict?

Suppose instead that execution begins, across all lanes, with testing at each subsequent stage of execution to see whether the predication value for a lane is true or false. Once we learn the value of the predicate (say in cycle three of six execution cycles, either we keep going [case true] or we shut down this lane for the remaining three cycles [case false].)

This speculation is feasible in the case of predication because, as long as you don't write back the value calculated on a false predicate lane, there's no harm in doing the work of the lane, you just waste a little energy.

Given this idea, the actual patent is various technical details about how this is implemented so that the final known predicate value can be used as soon as possible (so that we can stop wasting energy, for the false predicate lanes, as soon as possible).

We will see in various ways that this idea is still retained many years later; however as we get smarter in scheduling instructions, and have a larger pool of instructions from which to schedule, we will obviously try, when feasible, to schedule a non-speculative instruction.

instructions that perform free format conversion

Along with instructions, a few of which we have described above, there are also “instructions”, namely ways that data can be modified as input or output from some other instruction.

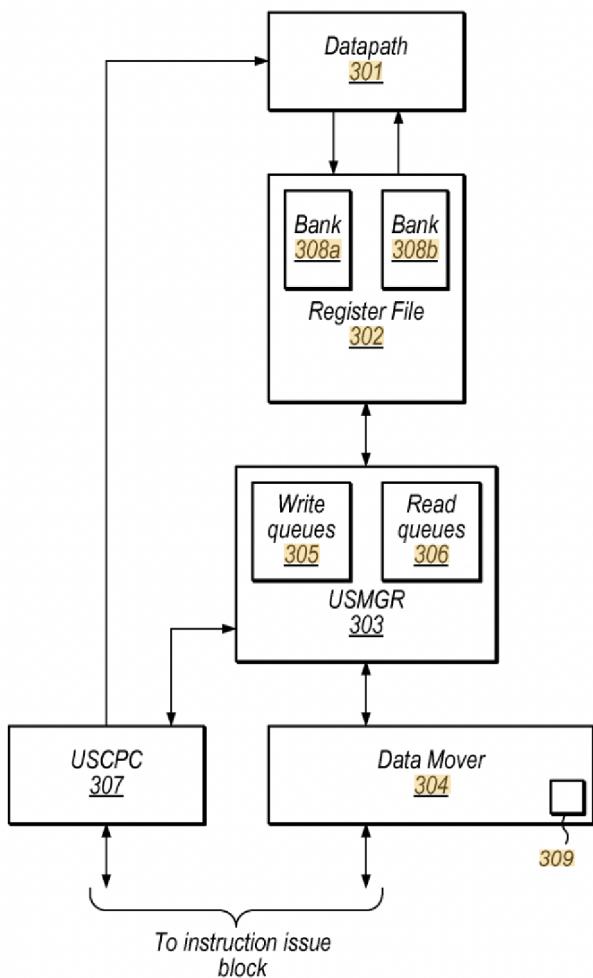
Some of these are fairly easy and obvious, for example many instructions have a few bits that indicate the final result should be negated, or the absolute value taken (or both, negative of the absolute value), or saturated (ie clipped to some maximum range rather than wrapped around, or treated as infinite).

But others are more interesting. For example the load and store instructions (and I think also the texture processing instructions) can transform data in a variety of ways, from fairly simple (zero extend values) to substantial data unpacking and re-ordering (eg load a packed bit format like rgb10a2 and

unpack it to 4 16bit RGBA values).

The set of transformations has grown over the years, but we can get an idea of how this is done from the early patent (2014) <https://patents.google.com/patent/US20160093014A1/> *Data alignment and formatting for graphics processing unit.*

Consider the diagram below.



The part that's important (and not obvious) is that the center of attention is that Register File which has two clients.

The high priority client is Datapath which (as I have said before) means essentially the FP/arithmetic units.

The lower priority client is USMGR (Unified Store Manager) which arbitrates Register File access for everyone else (load, store, texture, vertex, etc). USMGR has a bunch of queues (separate read and write queues for every register bank of the Register File) and deposits requests (register read or register write) in a queue as these come in from load, store, texture, etc. Recall that each of these is running separate clauses of separate warps, so they are all reading and writing to separate registers, but the registers for these different warps all live in the same Register File SRAM.

Another way to think of this is that Datapath represents the low-latency client, while everyone else represents clients for whom latency is usually less important.

So the overall model is that Datapath has the potential, every cycle to read or write registers. But occasionally it will not need to access the full bandwidth of the Register File (for example because the register cache, to be discussed soon, can supply operands) and during periods when any of the banks of the Register File are available, the first transaction in the appropriate USMGR per-bank read or write queue can sneak in. Mostly this should result in everybody getting their registers as they need them; if any of the queues do get close to full, the USMGR will send a message to USCPC (Unified Store Pipeline Controller) which will essentially throttle instruction flow to Datapath for a few cycles to let the queues in USMGR drain to some extent.

Next point is that you will notice that while Datapath has a direct connection to the register file, the secondary clients' connections go through Data Mover. Data Mover handles arbitration/priority (every cycle requests are coming in from load, store, texture, etc) and routing data “sideways” to the L1 cache or Scratchpad.

A neat little optimization is that requests from these secondary units can indicate that a register will be reused soon, so that the register is preferentially delayed a little in USMGR queue and the subsequent request for the register can find it in this queue rather than in the Register File – not a real Operand Cache like we will see soon, but a cute optimization, like I suggested for the CPU using the Store Queue of the LSQ as something of an L0 cache.

Finally we get to the point of all this. As data flows through the Data Mover, either in or out, it flows through the block marked 309, called the Format Unit. The Format Unit takes in data in large chunks (up to 16B in size) and rearranges them according to the reformatting request attached to the relevant load/store/texture instruction. This can do things like reshaping pixel formats (ARGB to BGRA) or change channel widths (556 to 888, or 8888 to 12121212, or whatever).

Thus the end result is that these instructions can perform some degree of data un/packing and rearrangement from a format optimized for storage to a format optimized for calculation without costing any additional cycles.

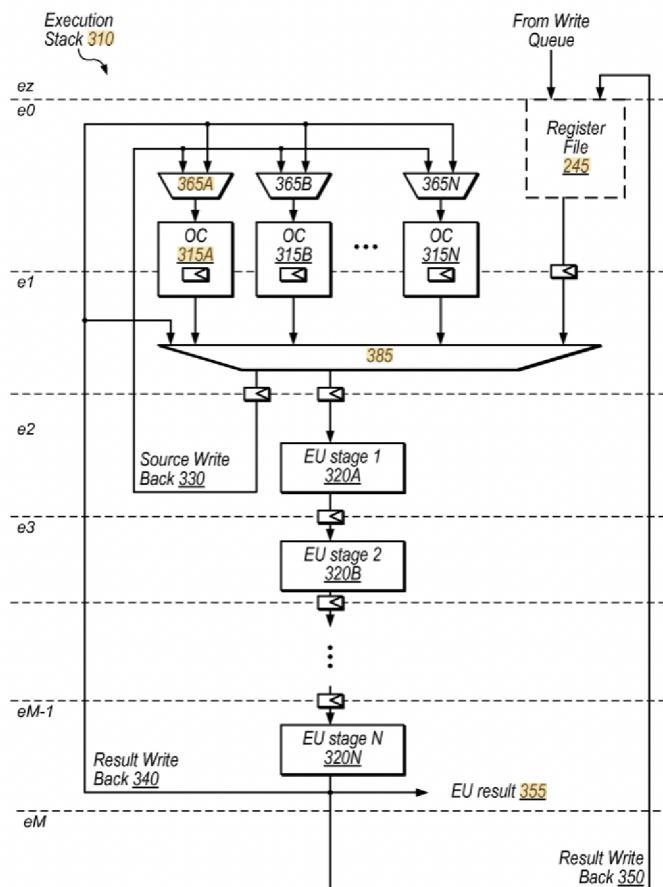
Later we will see that this two client model for the Register File goes away, we have a unified Read Queue and unified Write Queue for all clients, with more sophisticated arbitration/QoS. But presumably

“secondary” clients still flow through a Data Mover block which handles the sequencing of large (wider than a register) transfers, and the data reformatting.

non-pipelined FP

Here's a patent whose presence in current designs is unclear, but which raises some interesting issues. Look at (2014) <https://patents.google.com/patent/US20150205324A1> Clock routing techniques. Ostensibly this is about physical design and lower-level than we'd usually consider, but look at the following diagrams.

First we have



This seems simple enough. What's shown is a lane of a Datapath (ie an arithmetic) unit. We see that registers flow into the execution unit from a few different sources: the Register file, one or more Operand Caches, or Result Writeback (ie the value from the previous execution of this unit feeds into the unit again as source operand for a later instruction).

Already a few things are apparent:

- there are multiple operand caches. What's that about? I have no idea! However in the early days of operand caching, the operand cache was literally that, it was not a more generic "register cache". So there was one operand cache for operand A, a second one for operand B, a third one for operand C, and no movement between them. It was up to the compiler to try to optimize the choice of register A or B (using eg commutativity) to ensure maximal register reuse while in one of these caches. This doesn't seem to be an issue any more; when we get to the interesting operand cache patents, they don't seem to talk about separate caches for different (first vs second vs third) operands
- unlike the other operand caches of the time, Apple's cache has the ability to also cache result registers as we see, with some wiring to route the operand to any (perhaps more than one?) of the multiple operand source caches. Certainly nVidia did not cache results years ago, and in my looking through the changes each generation, I never saw a point at which they claimed to switch to caching results.
- there's a path for registers that don't get cached (presumably with some sort of permute/shift wiring) and, at this stage, the only way values get into the operand cache is by first being used as an operand, and then after use to be written to the operand cache.

Now think of the implications of this for latency. Suppose, to make up some numbers, the raw cycle time of an FMA operation is 5 cycles, and the usual register file access time is 3 cycles. (The register file is banked, there may be banking collisions, bad things are possible, but we'll assume the usual and very common case is 3 cycles).

So what is the latency of an FMA?

It might be 5 cycles if all the required operands are in operand cache, it might be 8 cycles if any of the operands has to be read from the register file. So how should we time the machine (eg in terms of compiler scheduling, and then in terms of hazard scheduling [instruction B has to wait at least N cycles after instruction A]) to handle this variable latency?

The easy option is to always assume the very worst case latency (which might even assume maximal bank collisions latency) and you can be sure that you are always safe to go.

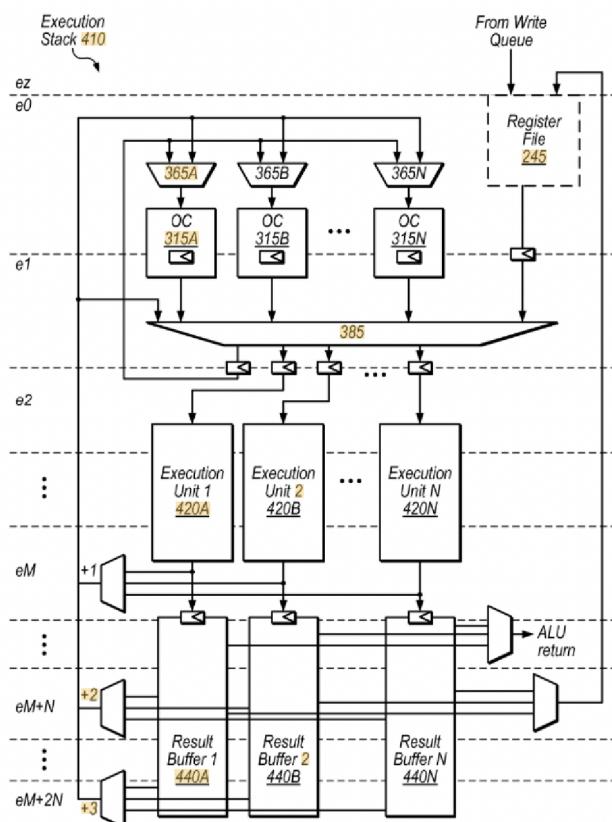
Obviously if bank collisions are very rare, but you are assuming worst case latency of say 12 cycles, now you are making it difficult to ensure a warp can execute every cycle. You might have a pool of say 6 active warps available for this quadrant, and while the compiler has tried to schedule sequences of independent instructions, there are limits to what it can do.

So next option is to assume that bank collisions are rare and schedule everything for, say, 8 cycles. Now you need a small amount of additional machinery to detect if all operands are valid, and to do something (like freeze the pipeline) if they are not all valid. When you freeze the pipeline you do waste a cycle or two that, in principle, might have been allocated to a different warp. So it takes some degree of simulation over a wide range of code to know whether it's better to suffer guaranteed delays (long latency, no warps with an executable instruction this cycle) or uncertain delays (shorter latency, but

sometimes bad luck in bank collisions reading from the register file).

A few years ago nVidia dropped their FMA latency to about CPU FMA latency, so 4..5 cycles. Apple, as of M1, had an FMA latency of ~8 cycles. As of M3 it's about 5 cycles. In both cases I think what happened is they moved to the next level of sophistication which is, if you trust your operand cache enough (once again, lots of simulations, plus larger, smarter operand caches) you can now schedule everything assuming operands always hit in the operand cache. Once again, if you guess incorrectly the clock for this execution unit will have to freeze for a few cycles while the operand values are collected, but once again it's worth it for the lower latency if clock freezing is a rare event.

That's all background; the primary interesting idea of this patent is below:



How is this different from the previous diagram?

The previous diagram assumed a pipelined execution unit. Notice that between each stage there is a latch which holds the value generated by the previous stage. That value moves on to the next stage in response to a clock signal.

The above diagram is unclocked; operands go into the unit, they flow through multiple gates over few cycles, and pop out the end as a valid result in cycle M.

If you don't clock the execution unit you gain a few things. You don't need a clocking signal for that logic (save power), you don't need latches between stages (saves energy) and you avoid the delay of moving data into and out of the latch, so timing is a little less demanding, meaning eg you can run your Datapath transistors at slightly lower voltage, again saving power.

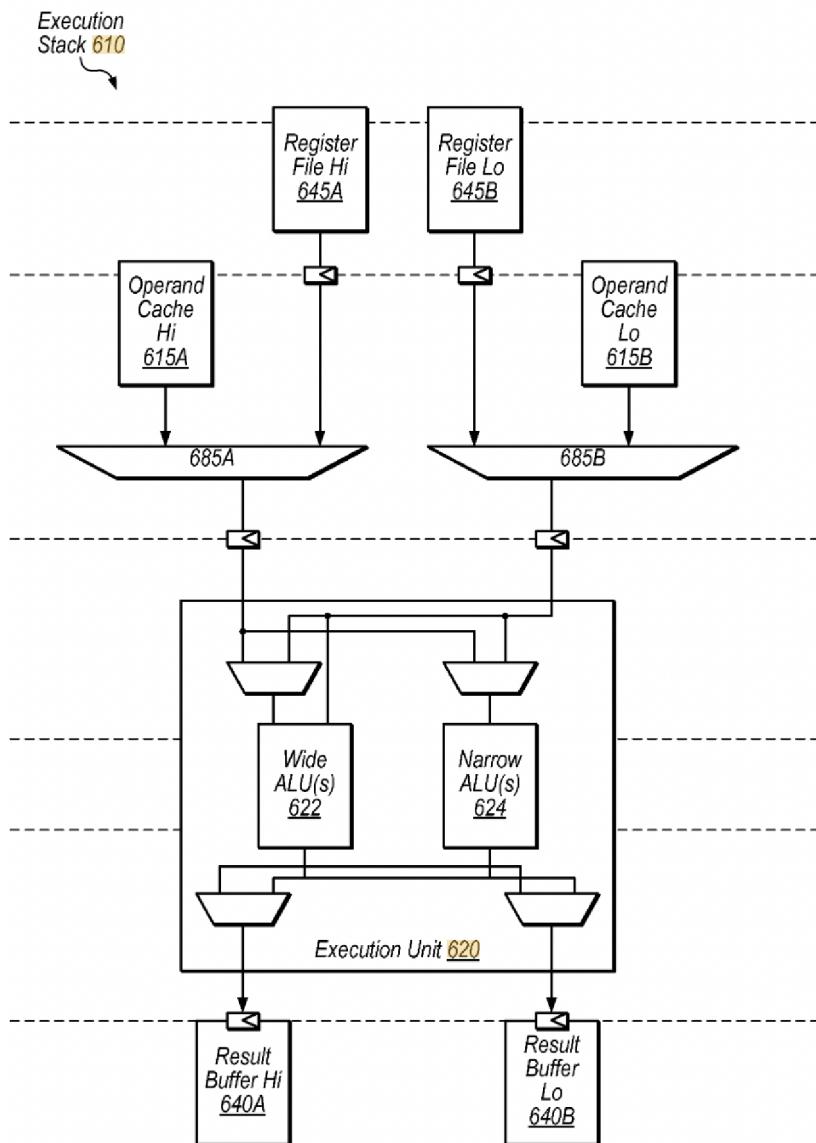
So why doesn't everyone do this everywhere? The big problem is that the execution unit is no longer pipelined, meaning that you can't feed it a new set of operands every cycle and have these flow through, each stage of multiple simultaneous instructions separated from the other stages by the latches. So if you still want the effect of being able to issue a new Datapath instruction every cycle, then you need multiple copies of the execution unit, as we see in the new diagram.

So the overall balance is

- we can save some power by using unclocked Datapath (no clock, lower voltages) BUT
- at the cost of more area. Not quite as much area as you might think because we save on clock wiring and latches, but definitely more area.

Does Apple still do this? Who knows? In principle you could even imagine that they design the Datapath execution unit at a high enough level of abstraction that they can use this unclocked version for say A and M/Pro series (where power is highest priority) and a clocked version (smaller, allowing them to pack in more GPU cores) for the Max and Ultra's.

Building on all this we arrive at:



Now what we have is an expectation that most data will be 16bits wide, but some will be 32 bits wide.

We store the high and low bits separately, and each cycle we decide:

- do we want low or high bits (full 32b register, or low or high half of the register) as each operand

- do we want to perform a 16b or 32b datapath operation

and based on these choices we can clock the bare minimum circuitry required for this cycle.

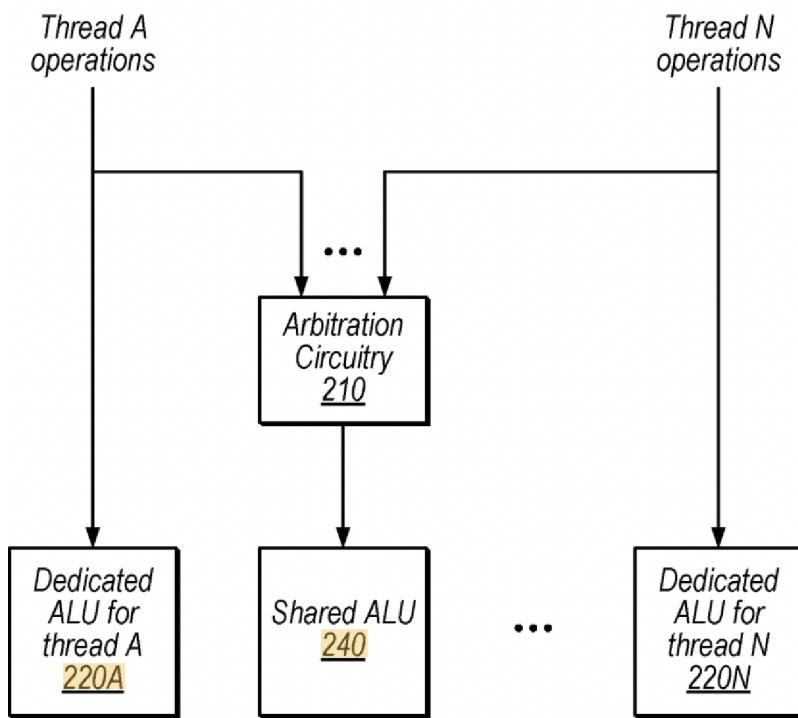
Note that we see here something to be discussed further, that rather than providing a single FP32 unit which can also perform FP16 operations, Apple provides an FP32 unit optimized for FP32 only operations, and likewise for FP16, again all in the name of power savings. (And again, if we assume no clocking, then both the FP32 and FP16 units are replicated a few times to allow for one instruction input per cycle.)

shared fp32 lanes

While the above patent, for simplicity, imagines an FP32 unit side-by-side with an FP16 unit, in fact up until the M1, the FP32 unit was shared by adjacent pairs of lanes, so that FP32 throughput was half that of FP16 throughput. That's still the case with the A14, but was changed for the M1 (and A15) so that they have identical FP32 and FP16 throughput.

(2018) <https://patents.google.com/patent/US10699366B1> *Techniques for ALU sharing between threads* confirms this.

(By thread below is obviously meant what I am calling lanes.)



One notable point (kinda obvious, but confirmed) is that physically the shared ALU is placed between lanes, as opposed to other alternatives like a block of 32 FP16 lanes situated next to a block of 16 FP32 lanes.

A second notable point is that a deliberate attempt is made to round-robin the arbitration of each lane for the shared hardware (the same would also be true for special-function shared hardware) rather

than alternatives like, say, randomly determining who wins.

The reason is: if round-robin is applied then, even when a lane has back-to-back dependent FP32 operations, the maximum amount of time (eg two cycles) will pass before the lane again has access to the shared unit, and, with luck, by that time the result will be available and so the back-to-back operation will not result in a pipeline stall. A random arbitration will occasionally result in a single cycle back-to-back execution for a particular lane, which will require a forced delay to wait for the result.

If Apple ever cared about GPU FP64 throughput, they could do something like share an FP64 unit with 16 lanes, which would give throughput comparable to e8m48 performance (a way of calculating something close to FP64 accuracy using FP32 in about 18 cycles), but easier to code. This could then be bumped up to eg sharing with 8, then 4, lanes, if it becomes used by important code.

An alternative, however, is that Apple believes the appropriate solution for FP64 is AMX, and they may be correct! This may be fast enough for FP64 purposes, and a much simpler and more natural programming model for the sort of STEM code that cares about FP64.

fp16 ↔ fp32 interconversion

As should be widely known, Apple really wants you to perform as much work as possible using FP16, and to this end, they make it essentially free to use FP16 data as FP32 and vice versa in almost all contexts. For example you can, without cost, load FP16 data to be stored as FP32 in registers and for computation, then write the result back as FP16. This may be good enough (do high precision computation, but deliver a lower precision result) and will halve memory bandwidth. But maybe you can go even further, load FP16's to generate FP32 intermediates and save some register space, and similar permutations.

We can imagine how the load/store conversions are done as data flows through the Data Mover Format Unit (changing from FP32 to FP16 is, more or less, a question of removing some bits in the exponent and mantissa, or replacing them with zeros, with a little extra circuitry, if you want to be finicky, to handle things like overflow; and recall that as of A15/M2 onward, BF16 will exist alongside FP16, but this just means a slightly different set mantissa vs exponent bits are removed/added).

When we wish to perform computations often the same sort of thing can be done simply in the datapath from the register file/operand cache to the execution unit, but there are a few cases that are slightly trickier. The most complex of these is when we want to calculate with FP32 inputs to generate an FP16 output. The obvious way to do this (to get the rounding and overflow details exactly correct) is to perform the full FP32 operation to generate an FP32 result, then in the next cycle perform an exact FP32→FP16 down conversion.

But, for whatever reason, Apple was unsatisfied with this solution. (2016) <https://patents.google.com/patent/US10282169B2> *Floating-point multiply-add with down-conversion* points out that the expensive part of the process is the same in both cases, and it's just a case of a few small changes in the final rounding/overflow detection step that differ between a 32b result and the 16b result (and which

you mostly need anyway in a separate 32b \rightarrow 16b rounding stage), so you can in fact place two parallel rounding/overflow paths, one to 32b, one to 16b, side by side; choose one at execution time, and save a cycle while not spending much additional area.

We see a slightly different idea here: (2017) <https://patents.google.com/patent/US20180121199A1> *Fused Multiply-Add that Accepts Sources at a First Precision and Generates Results at a Second Precision* which I'm placing here because I don't have a better idea where to put it.

You may know that nVidia uses a format named TF32 for its tensor cores. TF32 takes in matrices that are some variant of FP16 and multiplies them, but performs the accumulation (ie summing the results of the various multiplications) in a wider format than FP16.

This patent describes the same basic idea. Suppose I want to calculate $a = b * c + d$, where b , c , and d are all FP16. The multiplication $b * c$ will need to be done with more intermediate bits than FP16, and if I retain those intermediate bits for the addition of the $+d$, then I have a *fused multiply-add*, with some additional precision. So far, so normal, but for this normal case the system would usually then return d as an FP16.

What if I now want to perform sequence of such multiply-add's, as in, for example, a dot-product. In that case I might want to retain the intermediate sums as a higher precision throughout the accumulation. This is essentially what nV does with TF32, and is what Apple suggests in this patent.

The weird thing is that I don't know where Apple uses it. It doesn't seem relevant to the GPU or AMX. It's written as though it's for NEON, but I'm unaware of a NEON instruction that behaves this way. My best guess is that it's used inside the ANE, for the same reasons as nVidia.

fp32 power savings

If you can't get enough floating point implementation details, look at (2017) <https://patents.google.com/patent/US10481869B1> *Multi-path fused multiply-add with power control*. This patent (generically understood) is applicable, and probably applied, to the GPU, NEON, and AMX, at 16, 32, and 64 bit lengths.

Consider the operation AxB+C. In the most general case you perform the AxB, then you, depending on the exponents of this product and C, have to shift the mantissa of the product and the mantissa of C to line up mantissa bits and perform the sum, then there's some final cleanup to normalize the result, round the result etc. This shifting to align the mantissa's is energy expensive.

But if you are adding, say, a very large C to a very small product, this work may not even be necessary! The product may not change the value C, or may simply be the equivalent of adding or subtracting one bit.

This suggests that if you detect this case, you can route the sum down a different, cheaper, path that doesn't require this alignment, or at least can utilize less alignment work.

It's then a question of how many such different cases you wish to detect (for example another obvious case is when the product is very much larger than C), and how you detect such cases (most easily by comparing the exponent of the product to the exponent of C).

The patent suggests, with full details, considering the "full" case (also called the "near" case) and three

different unusual (also called “far” cases) that can each be handled in a way that uses much less energy.

compare circuitry

Another cute idea can be seen in (2016) <https://patents.google.com/patent/US9846579B1> *Unified integer and floating-point compare circuitry*. Suppose that you want to compare (less than, equal, greater than) two values. More or less (int or FP) that boils down to subtracting the two and looking at the sign of the result; and this seems like it implies the latency of such a subtraction (especially nasty for FP).

The patent points out that in fact you can do a lot better. You don’t actually care about the result of a real subtraction, all you need is a “subtraction-like” operation which delivers the correct value as far as $>$, $=$, $<$ is concerned. If you line up values correctly (whether INT or FP, 16 bits or 32 bits wide), pad some locations with zero’s and run the sign bits through some logic, the end results is that you can subtract the two values from each other (even two FP values) as integers in a single unified “comparator” unit and get the correct comparison!

special functions (pow function)

(2020) <https://patents.google.com/patent/US11372621B2> *Circuitry for floating-point power function* is a nice patent because it’s an idea I have never seen mentioned before. Consider wanting to calculate $\text{pow}(x, y) \equiv x^y$. While you will usually implement this as a library function, the problem with doing that is that before you can get to the real computation you have to deal with a *long* list of special cases

Case	Input Pattern <i>pow(x,y)</i>	Expected Result
0	<i>pow(+0,y)</i>	Returns $+\infty$ for y an odd integer < 0
1	<i>pow(-0,y)</i>	Returns $-\infty$ for y an odd integer < 0
2	<i>pow(± 0,y)</i>	Returns $+\infty$ for $y < 0$, finite, and not an odd integer
3	<i>pow(± 0,$-\infty$)</i>	Returns $+\infty$
4	<i>pow(+0,y)</i>	Returns $+0$ for y an odd integer > 0
5	<i>pow(-0,y)</i>	Returns -0 for y an odd integer > 0
6	<i>pow(± 0,y)</i>	Returns $+0$ for $y > 0$ and not an odd integer
7	<i>pow(-1,$\pm \infty$)</i>	Returns 1
8	<i>pow(+1,y)</i>	Returns 1 for any y , even a NaN
9	<i>pow(x,± 0)</i>	Returns 1 for any y , even a NaN
10	<i>pow(x,y)</i>	Returns a NaN for finite $x < 0$ and finite non-integer y
11	<i>pow(x,-∞)</i>	Returns $+\infty$ for $ x < 1$
12	<i>pow(x,-∞)</i>	Returns $+0$ for $ x > 1$
13	<i>pow(x,+∞)</i>	Returns $+0$ for $ x < 1$
14	<i>pow(x,+∞)</i>	Returns $+\infty$ for $ x > 1$
15	<i>pow(-∞,y)</i>	Returns -0 for y an odd integer < 0
16	<i>pow(-∞,y)</i>	Returns $+0$ for $y < 0$ and not an odd integer
17	<i>pow(-∞,y)</i>	Returns $-\infty$ for y an odd integer > 0
18	<i>pow(-∞,y)</i>	Returns $+\infty$ for $y > 0$ and not an odd integer
19	<i>pow(+∞,y)</i>	Returns $+0$ for $y < 0$
20	<i>pow(+∞,y)</i>	Returns $+\infty$ for $y > 0$
21	<i>pow(x,NaN)</i>	Returns +NaN when $x=1$ and rational or x is infinity
22	<i>pow(NaN,y)</i>	Returns +NaN when $y=\pm 0$
23	<i>pow(NaN.NaN)</i>	Returns +NaN

and if you implement the library function in the obvious way, testing for every one of these rare special case tests will take time (and I-cache space/bandwidth).

How can we avoid this? The idea is to have an instruction that performs all those tests in parallel and returns the appropriate result or that the instruction should proceed to the “real” computation. Neat!

Obviously this idea could in principle be used for other special functions when it makes sense, and, while the patent was submitted by the GPU group, I’d hope the CPU team is aware of the idea and also implements it...

(After I wrote the above, I read the Itanium floating point book: <https://www.amazon.com/IA-64-Elementary-Functions-Hewlett-Packard-Professional/dp/0130183482> which points out this very idea, that many special functions can be accelerated by a single “lookup” type instruction that handles all the weird cases in one cycle and then either bails or moves on to the “real” computation. Every good idea has been thought by someone already!

However thinking the idea is one thing, implementing it is another. Itanium, for example, does not appear to have implemented this idea much, at least not for the version the book describes.)

gate instruction

One way to make a GPU faster is simply to use time that would otherwise be unoccupied. This is seen with (2020) <https://patents.google.com/patent/US11204774B1> *Thread-group-scoped gate instruction.*

It's not uncommon, especially at the very end of a threadgroup, for only one thread (or perhaps one warp) to have to perform one final round of scalar or close-to scalar cleanup/summarization work. Wouldn't it be nice if the first (and *only the first*) warp that finishes execution (eg because it received its data from DRAM first) could go on to perform that final utility work? What we need is a way, across all warps executing a threadgroup, to say "if I'm the first to reach this point, continue with the cleanup code; otherwise do nothing". Now the cleanup can be performed at the same time that all the other warps in the threadgroup are doing their last computations.

The problem is that a normal barrier requires all the SIMD groups to reach a point, and we want the opposite of that!

The patent describes a new type of branch that is essentially `if (first_to_complete) { do the following work }` to take advantage of this situation. This is the essential idea, but there are a few tweaks and additional variants. For example we can allow the first N SIMD groups to perform the additional work, not just the first one; and we can even use this repeatedly during a computation if for some reason it makes sense. Examples might include checkpointing, or reporting out to a UI progress on a large computation?

This seems like a useful idea, but I can see no way, within the public version of Metal Shading Language to use it! The MSL 3.1 PDF describes various synchronization tools but not this one. Maybe this is a casualty of having to continue supporting Intel? Apple plans to make the functionality available in a future Apple Silicon-only MSL, and until then it's available purely for internal usage like the graphics firmware and Metal Performance Shaders?

Going forward, it seems to me there is value in aggressively generalizing this, as an instruction, and as functionality in Metal to allow a block of code to be marked as running in any (but only one of)

- the first lane of a warp that hits the instruction and/or
- the first warp of a threadblock that hits the instruction and/or
- the first threadblock of a kernel that hits the instruction

AND

the reverse of the problem (for the same sort of reason) namely execute as

- the last lane of a warp that hits the instruction and/or
- the last warp of a threadblock that hits the instruction and/or
- the last threadblock of a kernel that hits the instruction.

One could imagine Metal functionality, something like `if (firstToExecute [kLane | kWarp | kThreadblock]) { . . . }` where any or all of the three constants can be applied, and likewise for

`lastToExecute`. These could be compatible across all devices, implemented in the obvious way on older GPUs, but capable of using modern instructions on newer devices. Like the system supporting warps that are narrower than 32-wide, or providing a wide range of addressing possibilities (laneID, index within the kernel, index within the threadblock, etc) a way for the system to make developers more productive by handling the overhead of a common situation, which can then be hardware accelerated.

Here's another way to look at this.

The traditional GPU barrier performs two different jobs

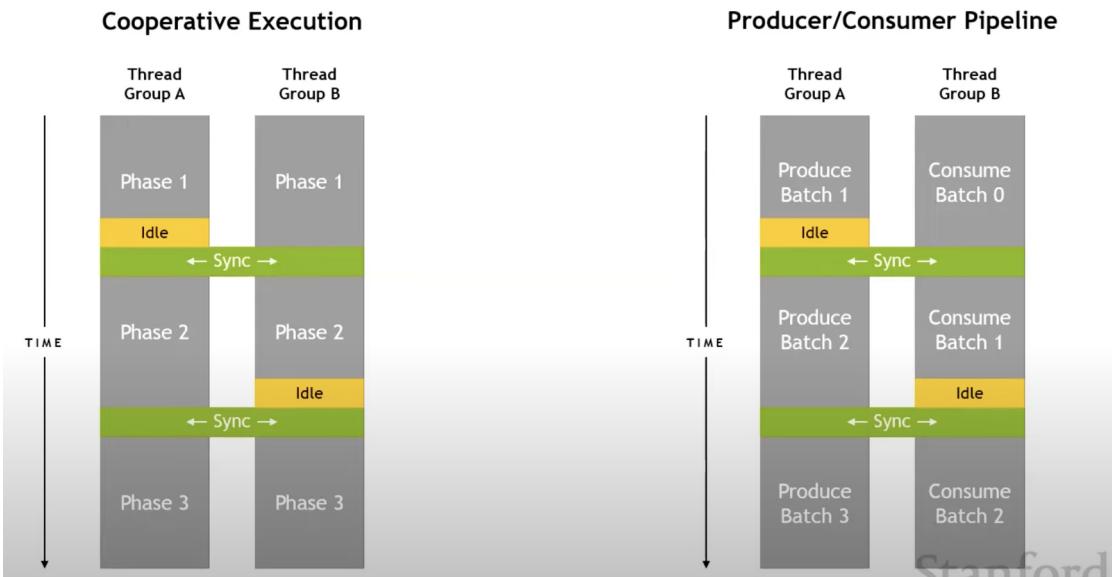
- it says *I* have reached this point AND
- it says I want to wait at this point for every other thread.

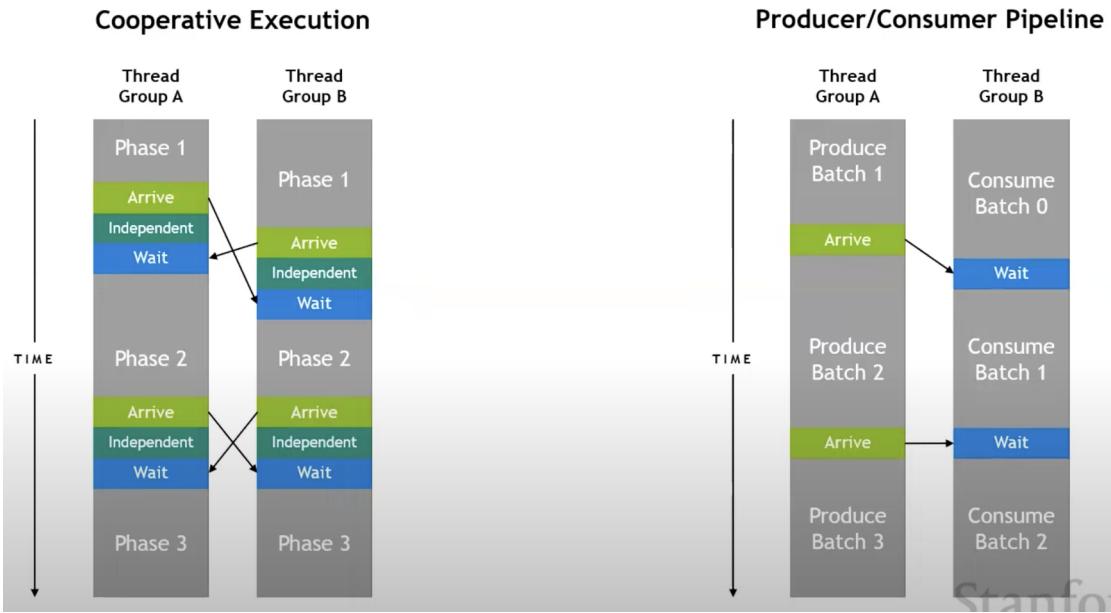
As we have seen repeatedly, one of Apple's design principles is to split things that do two jobs into two better-performing single jobs. The gate instruction is a version of this. But you can do better and, no surprise, nVidia does so. The Ampere architecture provides barriers split into an "Arrive" part that performs the first task, and a "Wait" part that performs the second task.

More formally "Arrive" means that a thread/group has finished producing (eg writing to) some shared data; while "Wait" means a thread/group is ready to begin consuming (eg reading from) some shared data.

AMD has also added split barriers, as on 2025's RDNA4 architecture. AMD calls the two halves `barrier_signal` and `barrier_wait`, which seem to me the nicest names by far in terms of making it clear exactly we're trying to do, but for the diagram below we're stuck with the formal names.

Their presentation at Stanford, (2023) <https://www.youtube.com/watch?v=MC223HlPdK0>, *Stanford Seminar - Nvidia's H100 GPU*, includes the following two diagrams:





Consider the left hand side first. We have two thread groups working together. Under the standard barrier model, when one of the thread groups completes, it has to perform a joint “publish-that-I-have-arrived-AND-wait” barrier and spin doing nothing (the yellow idle time) until the second thread group executes that same “publish-that-I-have-arrived-AND-wait” barrier and they both move on. Under the new barrier model, the first thread group can publish that it has arrived (ie I have done what the second thread group is waiting for) and then do whatever independent work it can find. It only performs a wait when it has finished doing its independent work, and that wait only lasts until the second thread group indicates that the second thread group has arrived (ie done the joint work that anyone else might be working on). The lower left diagram looks a bit complicated, but it makes sense when you think about.

The right hand shows an even stronger version of this. Now we don't require any “joint” synchronization, all we need is that the producer thread group be able to indicate to the consumer thread group

that a new batch of work is available, and all we need is for the consumer thread group to wait on that batch of work. Each side only in fact needs one half of the split barrier (either the Arrive side or the Wait side) so even the basic operation of synchronization can be cheaper, since less information has to flow between the two thread groups.

As of Hopper, nVidia both makes the above design more efficient (some details of how the Wait is performed) and allows Waits on pending memory transactions. Exploring this would take us too far afield, let's just say it deals with the issues of Acquire/Release that we have already discussed with CPUs, and allows a consumer waiting on data to be sure, once it passes the execution synchronization barrier, that memory is also synchronized, without requiring the additional Acquire/Release barriers and a separate flag that are normally required.

but essentially it allows a single barrier now not just to synchronize say a consumer with a producer in terms of where they are in execution, but means at the end of that same synchronization the consumer can be sure that all data of interest is stored in a readable location

So to summarize, what we have in the scoped gate instruction is Apple realizing a part of the same insight as nVidia, but each are implementing the insight in a different way, at a different level (either the SIMD or the thread group level). Neither seems yet to have thought through the implications of providing a *full* set of split barriers at every level from SIMD to thread group to GPU/CPU synchronization...

other

There are other interesting instructions, for example those that implement permuting and matrix multiplication; but before we get to them, let's discuss the operand cache.

The Register File and Operand Cache

What's called the register file for a GPU has to be massive because it contains

- so many registers (128 registers each 4B long)
- across so many lanes (32 per core quadrant)
- across so many distinct threads (at a minimum 3 if they each use all the registers).

Recall that Apple provides a single warp with 128 4B registers which can also be treated as 256 2B registers, and that the pool of registers is shared across warps so that if a warp uses fewer than 128 registers, it may be possible to pack more warps onto the core, leading to higher throughput (fewer cycles when we're unable to find a runnable thread).

This size means that the register file is ~200kB in size and is actually implemented in SRAM rather than as a register file, with the consequent constraints on simultaneous access.

We work around this in the usual ways

- with multiple SRAM banks, so that in any given cycle (with luck) we can direct multiple different accesses to different banks

- by caching operands in an Operand Cache which is a cache-like structure but implemented in register, not SRAM, technology, and thus capable of low latency (and, if we wish, multi-ported) access.

Let's start by discussing the register file.

The Register File

As usual, it's easiest to start by thinking of a single lane. Assume we want (on average) to be able to perform one read and one write per cycle, this means we need (at least) two banks. One bank (so one read or one write per cycle) is probably too low, even with the operand cache. The obvious way to do this is have the even numbered registers in one bank, the odd in the other bank.

The implications (eg if I want to read two odd numbered operands which are both not in the operand cache) are clear, but hopefully caching, prefetching, and a write queue limit the performance impact.

Note that if you only have two banks then, on average, half the cycles that have two requests (a read and a write) will collide...

And maybe you'd like to be able to service at least two read per cycle (since FMA's have three inputs). So perhaps we'd prefer to have at least four banks.

Next note that this calculation assumes four independent register files, one for each quadrant. The design probably started off that way, but at some point seems to have switched to a unified register SRAM, and we will see there are good reasons to assume this must be the case for the M3. If you have a unified SRAM serving four quadrants (ie four sets of client execution units each cycle) now you probably want at least sixteen banks.

Next realize that it seems to makes sense to treat the entire 32-lane-wide elements of a register as a single object, a "register" of $32 \times 4B = 128B$ in length because that means only one set of control signals to read the entire register and move it (and break it up) to each lane for use.

However this is sub-optimal in terms of power because you don't want to be reading (or writing) register values that are predicated out. My guess (though the patents do not discuss this) is that Apple distributes per-lane control signals as close to the SRAM as possible (ideally all the way to the sense amps) to limit read/write of predicated-out registers.

4-lane-wide SRAMs

So in principle we could simply have two banks of registers each say 128B ($32 \times 4B$ values) wide, a value for each lane. But the physics of SRAMs, and energy concerns, make that sub-optimal.

In fact the unit of width seems to be 4 to 8 lanes wide, so in a bank the slot for register 0 has four entries, for lane 0, 1, 2, 3; then another bank has four entries for lanes 4, 5, 6, 7; etc.

Imagine something like this, taken from a 2013 patent:

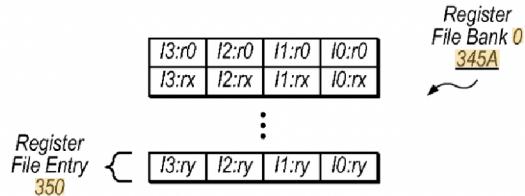


FIG. 3A

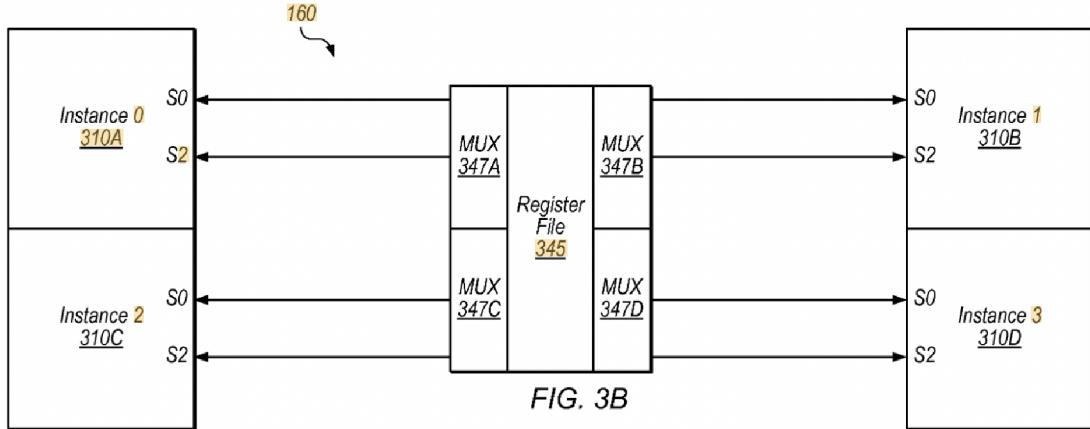


FIG. 3B

We have a physical layout of four lanes, two on each side, with the register file in between them. So physically the register file is split into 8 separate SRAMs, one for each block of four lanes. (And, at least as of 2013...) this makes it clear that SRAM is very much physically per quadrant (and in fact per set of four lanes) not a single pool of SRAM across the entire core.

This set of four lanes is called a quad (not to be confused with a quadrant, a quarter of a GPU core incorporating, among other things, 32 lanes).

So within this pool of SRAM for four lanes, we have say 3(warps)×128 slots, one slot for each register 32b number (and 3 sets for at least 3 active warps), giving 384 slots.

A single slot is let's say 4(lanes)×4B wide, and slots are distributed over let's say two banks, one for even register numbers, one for odd register numbers.

So the obvious consequences of this include:

- a single lookup in the SRAM sees the four (32b) registers for these four lanes. If 16b registers are desired, we can extract the relevant low or high bytes from the lookup.
- we can read or write one even and one odd set of four registers per cycle, but otherwise will have a bank collision
- if we know a set of four lanes of a quad are predicated out, we can avoid register access in that case
- in the obvious ways we could vary this scheme, for example using four banks rather than two, if that made sense
- it might not be too hard to shuffle the data that's delivered to each lane, so that some degree of permuting is possible within these 4 lanes. That's useful for some purposes – but it is only within 4 lanes. So stay tuned...

optimized Write Queue arbitration

Much of (2013) <https://patents.google.com/patent/US9330432B2> *Queuing system for register file access* is surely obsolete. But there are interesting elements that are likely not obsolete.

Recall that we split access to the register file into a low latency client (Datapath, aka FP/ALU) and multiple other more latency tolerant clients (load, store, and various graphics hardware like texture, vertex processing, and fragment generation). These other clients all go through a read queue and write queue, to access the Register File when it's not in use by Datapath.

Apart from arbitration to access the Register File (Datapath has high priority, read queue second priority, write queue third priority) there is a second level of arbitration to access the Write Queue, since in any cycle more than one client may wish to perform a write. Here's where things get interesting!

Many of these non-Datapath clients have the characteristic that they perform “wide” writes. For example the Load unit performs possibly wide(eg four 4B values) reads, which then have to be written to the register file; or the texture unit may generate a 2×2 block of 32B (RGBA) color. So think what this means for the write queue.

Conceptually at least, you can imagine the queue as slots that are, say, $32 \times 2B$ wide (sometimes we may have the minimal writeback of a single register 16B result from Datapath), but we have many of these slots, and some operations like texture or a wide Load will need to fill in many (eg 8 or so) of these slots. So we will have a situation where a single unit, eg texture, starts writing to Write Queue slots, but will have to perform the writes to multiple entries over multiple cycles.

We also have a situation where some clients (eg Loads that all hit in L1D) can write the entire width of a queue slot in one cycle.

Other clients, like texture, generate one block of texture per operation, and so may write partially (to the lanes of only some registers, eg $4=2 \times 2$ registers) in each write request. But a texture unit writes multiple colors to each pixel, so it writes to multiple registers, so a texture write, in one transaction wants to

- write to eg four registers (one for R, G, B and A) but only to
- four of the 32 lanes of those registers.

You may get something similar for Loads, especially if the Data Mover has reordered the load data, but now it will look more like in one transaction the Data Mover wants to

- write all the R elements (one register) for 32 lanes, then
- next it will write all the G elements across 32 lanes, and so on on

This all has implications for the arbitration of access to the write queue.

- Suppose we delay the write out of textures. Then, while those are delayed, some slots in the write queue are blocked, unavailable for use by other writing clients, but also unavailable to be written to the Register File
- Suppose we delay the write out from the Data Mover. This will result in a full slot of 32 lanes worth of values in the write queue (which can then be moved to the Register File as opportunity arises); but it will halt the progress of the Data Mover, which can't move forward until it has fully dumped to the Write Queue the current set of data rearrangement it is working on.

Thus we see that there are optimal orderings for arbitration to the write queue. Insofar as possible, once we start a multi-slot transaction we want to prioritize on-going requests to partial slots (highest priority) or full slots, but more than one of them (secondary priority) above other requests. Another, simplified, way you can think of this is as prioritizing access to the write queue based on whether a transfer is one cycle, or the first of two cycles or the second of two cycles. (Basically you want to handle second of two cycles at highest priority because it will clear two entries in the queue; give medium priority to one cycle transfers because they will only use one queue slot; and give lowest priority to first of two partial cycles, because they "clog up" the queue while you wait for the second transfer. If you delay this first transfer while you handle higher priority transfers, simultaneously the TPU can calculate the next set of values, so that ideally very soon after you perform this transfer, the second set of texture pixels, filling in the rest of the partially filled write queue slots, will be available.)

This is as opposed to alternative arbitration schemes you could imagine like round-robin or oldest (which we might use for the baseline case before any multi-slot transactions have begun).

use of Write Queue as an “operand cache”

A further attempt to extract value from the write queues can be seen in (2014) <https://patents.google.com/patent/US9437172B2> *High-speed low-power access to register files*. We've already discussed this, but now we will do so in more detail.

Now we give each write stored in the Register File Write Queue a flag that marks the write as "hold". Writes marked as "hold" are considered likely to be reused soon, so that it's energetically cheaper to read them from this write queue than from the register file (along with not using Register File bandwidth). This is like using the Write Queue as a cheap Operand Cache for the non-Datapath clients.

For example imagine some code takes the form of loading some values into registers, then executing some Texture Processing instructions. The compiler can notice this and mark the loads as Hold; so they will flow through the Data Mover into the Write Queue; then ideally sit there until the texture unit clause starts executing (after the load clause has finished and indicated all operations complete). The

texture clause will try to read those register values and, with luck, they can be extracted from the Write Queue with no Register File access.

Values are held in the write queue for as long as possible, only being written out to the Register File (oldest first) when the Write Queue gets close to full.

The other notable point in this patent is that every time the Read Queue for each bank is unable to perform a read, a counter is incremented (and then reset if the next cycle a read is performed). The net result is that, at any given time, we can see that, for example, Bank 3 has not serviced any Read Queue requests for 10 cycles. This allows us to define a concept of a bank being *victimized* ie unable to service read requests for longer than some number of cycles.

At this point the patent provides an interesting example of: define the problem *exactly* and solve it *exactly*.

The problem is not something like “The Read Queue doesn’t have enough accesses” and the solution is not “stop Datapath for a few cycles”.

The problem is precisely:

- The *Read Queue* for *Bank N* has not had access for *M* cycles (ie this is a *latency* issue, not a bandwidth issue), and the solution is
- Prevent Datapath from accessing *Bank N* for a cycle. Don’t freeze Datapath completely, just put a hold signal in the line to *Bank N*, so that if the next Datapath instruction doesn’t need *Bank N* – hits in Operand Cache or another Bank – then it can go ahead just fine.

Note also that this is not how it works for Write Queue; Write Queue does not care about latency. Instead the main thing Write Queue cares about is that when a particular Queue is full then clients of that queue will have to halt, so it’s when a Write Queue becomes full that it raises the same sort of flag, and access to the Register File Bank by Datapath is suppressed for one cycle.

The Operand Cache

At this point we’ve seen enough to understand that the Register File is conceptually a serious bottleneck (latency for reading registers, bandwidth for both reading perhaps 3 register for one FMA operation along with writing back 1 register result, along with other clients like Load, Store, TPU, Vertex, Fragment, etc who also want a constant stream of access). And a cache of recently used registers seems like a good idea. But is it? How much reuse of recent registers actually exists? How large should such a cache be? How should it be managed?

It’s clear that this is an area where some serious simulation and academic work are necessary, because the answers to these questions are not obvious!

It’s worth thinking about how an operand cache might work. The most obvious, simple, idea, is that as we load each *source* operand for an instruction (remember these “register operands” are 32 lanes wide), we store it in a cache. Each time we need to load a source operand, we first look in the cache. We have various choices available.

How large should the cache be? Obviously we base this on simulations and how much power and area we want to provide. If the cache is small enough (eg 10 values, as suggested in some of the early papers, 32 values is suggested in early Apple patents) we might make it fully associative. But if it’s

larger then we might want to make it set associative, eg 32 elements arranged as 8 sets of 4 ways, the set chosen based on the lowest 3 bits of the registerID (or some similar 3-bit hash).

As we make our cache smarter (and thus achieving an ever-higher hit rate, we may be willing to devote more area to it, especially if cache hits are lower energy than accesses to the register file).

We then have to choose a replacement policy, anything from random to not-most-recently-used (which only requires a single MRU bit per entry) to LRU (which requires storing enough info to be able to order the entries by usage time).

A source-operand-only cache (ie a read-only cache) avoids having to worry about changed values, but still requires some careful thought; in particular when a destination register matches a register in the cache, we need to invalidate that cache entry.

The next step is to provide a path to allow results from Datapath to simultaneously write back both to the Register File and (if there is some reason to believe the value will be reused) the Operand Cache. This will at least avoid having to re-read from the Register File.

Most ambitious would be to only write back to the Operand Cache, which will allow temporaries to be overwritten while in the Operand Cache, never wasting energy writing to the Register File. This will require some mechanism (possibly a compiler hint) to ensure that any modified register in the Operand Cache is written back to the Register File before being used by some non-Datapath clause.

We can get a lot more value out of the Operand Cache if we are willing to extend the ISA to provide some flag bits attached to every operand. If we do this, what flag bits should we use?

The easiest and most obvious one is a “last use/kill/purge” flag, which the compiler will attach to a register that is not re-referenced for many cycles. By freeing the cache slot immediately rather than waiting for it to age out, we increase the “effective” size of our cache.

This same bit can actually mean “don’t cache” in the sense that if we use it on a first and only use then the caching system has enough info to not allocate an entry and just bypass the data straight from register file SRAM to instruction execution.

Depending on the cache replacement policy, we might also provide some sort of “hold” flag. For example we could have a policy that registers without a hold flag are subject to a random-notMRU policy (fairly easy to implement), so we get a tier of “temporary” registers that get replaced as needed and “higher priority registers” that are retained for a while. We will see this added in a later version of the cache.

We will also need a way to eventually replace these “hold” registers. Many options are possible in terms of balancing cache priority with easy implementation. For example you can do things like random replacement but, when you choose a random, to be replaced, register, if a “hold” register is chosen, then try again. Allow maybe three tries, if third attempt is still a hold register, tough, it gets replaced.

Less obvious is that we might also provide a “preload” flag. This seems kinda weird – won’t we immediately use the operands listed in the instruction?

The key insight is that the usual case is that an ISA provides only a few instruction formats, so that for example the same format used for an FMA instruction (which takes three source operands) may be

used for an ADD instruction (two source operands), or a NEGATE instruction (one source operand). Normally the instruction decoder would just ignore the unused operand fields, but the compiler can put valid register values in those slots, together with a “preload” flag. Then, with luck, the compiler can arrange things so that usually we have all our source registers already present in the cache with later-use registers smuggled into the cache via a preload bit attached to earlier instructions!

Another way you might deal with the issue of no longer useful cached registers is, at certain points, after performing one block of code (think of something like a function call) you might want to invalidate all the registers, or at least remove their “hold” flag so that they can easily be replaced because they’re mostly not relevant to the next block of code/function call. This is a different kind of flag, because it’s control of the entire cache, not of a particular register within the cache.

The next big step is if we’re willing to also store results in the operand cache. This now requires some care in terms of making sure that the cache does not get out of sync with the register file. Particular attention has to be paid to predicated instructions because now only some, but not all, lanes of the destination register will be written out.

Think about it: The obvious way to handle writes to the operand cache is just to overwrite the destination slot in the cache, without having to interact with the register file. But that’s not good enough for a partial write (as in a predicated result).

There are multiple ways to handle this, the most obvious being to first load the operand cache with the original register from the Register File, then overwrite the predicated lanes. (Even better we only load the non-predicated values from the Register File...)

We’ll see implementations (sometimes varying from an early design to a later design) of all these ideas and issues.

Academic Results

You can read a few papers to see what they think of Operand Caches. It’s worth noting that most of these appear a few years after Apple’s patents with the same ideas.

(2018) <https://www.pdl.cmu.edu/PDL-FTP/associated/ltrf-asplos18.pdf> *LTRF: Enabling High-Capacity Register Files for GPUs via Hardware/Software Cooperative Register Prefetching*, describes compiler-driven register prefetching, but which is probably not as practical as the Apple scheme in terms of easily retrofitting it into an existing design, or as widely applicable.

(2021) https://people.inf.ethz.ch/omutlu/pub/latency-tolerant-highly-concurrent-GPU-register-files_tocs20.pdf *Highly Concurrent Latency-Tolerant Register Files for GPUs* describes a scheme very close to what Apple does. The primary difference seeming to be that this publication adds a separate “register prefetch” instruction, rather than piggy-backing prefetch onto slots in existing instructions.

This second paper also has some numbers that make a few interesting broad points. Let’s list these along with a few points from other (unmentioned) papers

- it is still *usually* the case (at least for nVidia GPUs) that thread switching is not enough to fully hide DRAM latency. Meaning that across a range of code, about 30% to 50% of cycles are stall cycles

where no thread executes.

This could be improved by packing more thread blocks into a core, except that each thread block requires a new set of allocated registers, so we are constrained by the number of physical registers available.

Note that it is usually the number of physical registers that's the limiting factor, not other resources to be allocated like, eg, space in the local address space or in lane local (ie per lane stack space).

This means that we're always after a larger register file, but that in turn means a slower register file. Which in turn means ideas, like operand prefetching, to make the register file effectively faster are very valuable.

Note that it's unclear whether these nV numbers are as grim for Apple. nV uses HBM of course, which has great bandwidth but still has DRAM latencies, and nV have generally scaled up their designs with HBM bandwidth so they're still in this tough spot. Conversely Apple work very hard to capture much of their graphics working set in the SLC which is much lower latency than DRAM, and have 50% more registers, so can pack more warps waiting for memory onto a core.

It's also **important** to realize that that 30% to 50% of cycles doesn't exactly mean what most people think it means...

In particular, it's generally considered to mean that we must either increase DRAM bandwidth (eg HBM), or reduce its latency (eg SLC), or provide more threads. Those are all useful, but in fact (and rarely mentioned) a substantial contributor to this issue is in fact delays related to the TLB! So a fourth avenue of attack on the problem, one that has so far been inadequately explored, is a much better, GPU-optimized, TLB subsystem. I discuss this later in this document.

- simple-minded HW register caching (ie just LRU cache register values as they are accessed, in the hope of reuse, and with no hints) works badly, with ~30% hit rate in a practical-sized cache. Along with the obvious explanation – you don't get that much temporal locality – a second issue is that, unlike an L1D cache, you don't get the benefit of spatial locality ("nearby" registers with a slightly smaller or larger register number are not necessarily likely to be used soon along with this register, and weren't pulled in as part of the same "cache line"; unlike memory address space).

One reason for the low temporal reuse is the compiler wanting to schedule interleaved independent instructions, which means no waiting on the result of a previous instruction, but also means many more registers in "active" use (and so too many to be effectively cached)...

The situation is, however, dramatically improved if you use SW cache management

- to prefetch as required, and
- to mark registers as purgable, when the value can be dropped from the cache.

Unlike any of the patents, this article shows a diagram of how such a cache might be set up. Other designs somewhat conflate the Operand Collector with an Operand Cache or Register Cache.

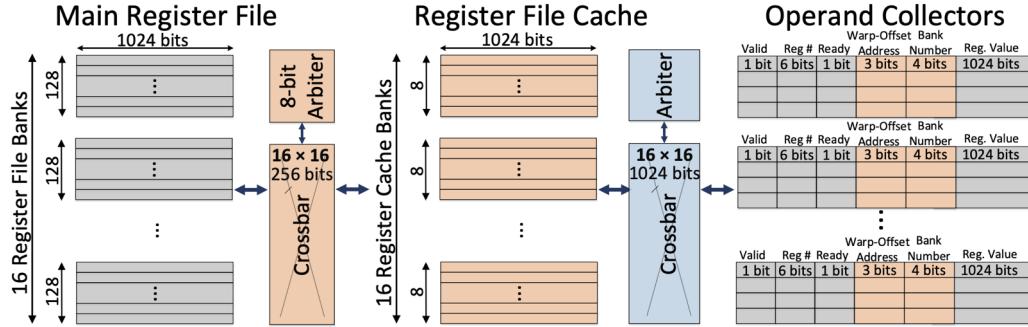


Fig. 11. LTRF architecture. Figure assumes 8 active warps, 256 architectural registers per warp, 16 register file (cache) banks, and 16 operand collectors.

It's a little unclear, at least to me, but the article seems to suggests that this operand caching by itself can improve performance by ~30% (again across a range of apps, with, as usual, some benefitting a lot some very little). But caching may improve performance even more via second order effects, in that more registers can be provided (ie a slower register SRAM is allowable) so that more warps can now be packed into each core, allowing for better DRAM latency tolerance.

Associated with operand caches, once you have a scheme that (hopefully) usually has the registers you require available when you need them, what's the optimal scheduling of instructions? Now we're not considering the scheduling of kernels, or threadblocks; we're assuming a certain set of warps has been placed on a core (or a quadrant) and from this pool of warps (say of order 5 to 20 warps), each cycle, we want to choose an instruction to execute.

The easiest GPU scheduling algorithm is LRR (Loose Round Robin) which implements round robin, skipping over threadblocks that for whatever reason are not able to execute that cycle; either because they depend on a not yet available result, or on a memory load.

This has many inefficiencies, the most obvious of which is that it tends to produce a thundering herd effect (all the threadblocks that want to access long-distance memory tend to land up executing together, so all hitting memory at once [bad] and becoming retired from execution [bad]).

Next up is TL (Two Level). This is like the earlier versions of what Apple was doing. Thread Blocks are segregated into two pools, an active pool and a pending pool (most of which are waiting for DRAM, some may be waiting because they haven't even started yet). We round robin across the active pool, but as soon as a pending pool element receives its data, we try as soon as possible to swap it in to the

active pool.

Done carefully, this usually does a good job of interleaving memory access and computation across the core. But it doesn't make optimal use of cache.

GTO (Greedy Then Oldest) is something of a hybrid of threadblock and warp scheduling: a single threadblock (usually multiple warps from that block) is executed for as long as possible, until all its executable threads need to move to the pending pool; then we try to do the same for the oldest threadblock.

This gives us all the advantages of TL, but also tries to get as much cache reuse as possible from the execution of a single threadblock. Obviously there is a balancing act here between trying to stick to a single threadblock (and then threadblocks within the same kernel) as much as possible (hoping for cache reuse), but also trying to interleave with different threadblocks (to get work spread across different types of execution units)... I suspect an optimal algorithm will, at the very least, be tracking what the critical resource is at any given point (are we mostly waiting for L1? For DRAM? For the texture unit?) and will toggle between sticking with a single kernel and alternating between different kernels depending on the circumstances.

The academic state of the art was something called CCWS (2012) <https://people.ece.ubc.ca/aamodt/papers/tgrogers.micro2012.pdf> *Cache-Conscious Wavefront Scheduling* which tries to get you all the advantages of GTO, but with explicit optimization for cache reuse, rather than this falling out of GTO as a happy side effect of the Greedy part of the algorithm. There may now be something better than CCWS?

I mention these not just so you know the vocabulary, but because we have simulated numbers for how much of a difference this scheduling algorithm makes (at least on a certain simple model of hardware).

GTO (for the subset of benchmarks that are primarily memory constrained...) is about twice the performance of LRR.

CCWS is about 1.5x the performance of GTO, and about 86% of theoretical optimum (ie an algorithm that with future knowledge, at every cycle, made the best possible choice for the next thread to execute).

This sounds good, but you always have to remember the caveat of "for the subset of benchmarks that are primarily memory constrained"; the actual algorithm we want Apple to implement has to do well across all workloads, not just memory constrained benchmarks. (In particular, real world use cases have a lot more of interleaving code that accesses different hardware; academic benchmarks tend to do one type of computation; so for Apple it's more important than for the academic benchmarks to balance cache reuse against starvation caused by using only one piece of hardware rather than distributing work across all the different execution units.)

What we have already seen with Apple is a concern (in the creation and scheduling of threadblocks) with reuse of data within the cache, so we can assume that part of things is handled. Apple's concerns appear to have been, as we will see,

- running multiple distinct execution units simultaneously. (Choice of which clauses from which warps to execute next.)
- limiting how long any instruction waits around before it can execute. (First operand cache, then prefetching into the operand cache, then scheduling to avoid waiting for an earlier dependency to be calculated, then scheduling from a pool of two “partially executed” warps, to choose the one that has the least waiting ahead of it.)
- balancing instructions that can issue once per cycle (to Int or FP) with those that can only issue every second or fourth cycle (eg special purpose instructions).

Apple's designs

initial 2012 design, based on evolving instruction bypass

(2012) <https://patents.google.com/patent/US9600288B1> *Result bypass cache*

We begin with a barrel processor and a large multi-banked SRAM register file, with consequent pressure on the SRAM (support multiple accesses per cycle, and accesses are energy expensive).

Our first improvement is to try to schedule instructions so that, as much as possible, one operand for the next instruction is generated by the previous instruction (and so can be read off the bypass bus rather than from the Register SRAM).

We can compare these two schemes below: in the second diagram we see a bypass path from result writeback to possible operand.

Note that by Operand Storage Area is meant the Register File SRAM.

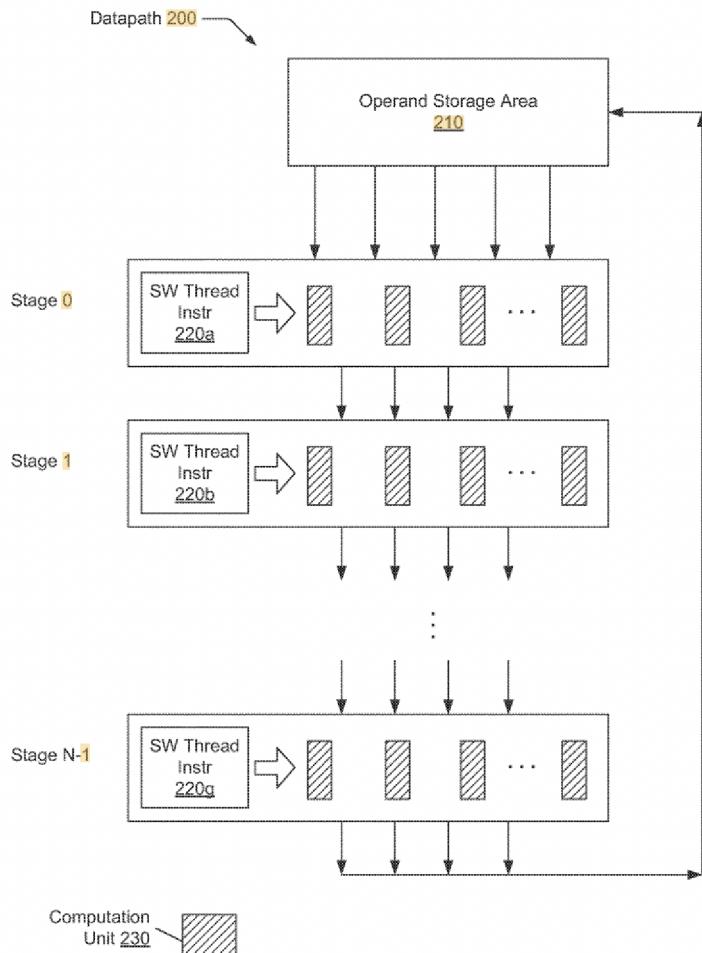


FIG. 2

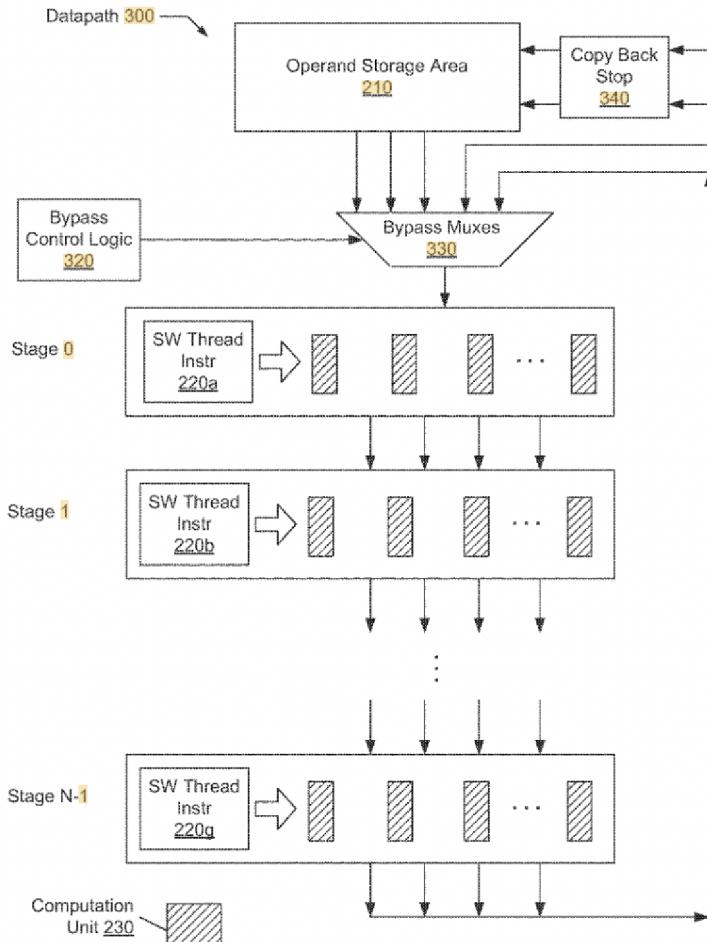


FIG. 3

In this design we include one hint, namely “dead value” or “last use”.

Suppose we generate a value, rTmp, which then goes into the next instruction and is never used again. We can hint, in this next instruction, that this is the last use of rTmp, and this hint will be used by the Copy Back Stop logic to prevent writeback. Note two points

- it makes sense to add this hint in the next instruction (as a “last read” hint) rather than in the previous instruction (as a “one more read use” hint) because it’s the next instruction that will be controlling

bypass and can easily, at the same time, assert the Copy Back Stop signal.

- a “last read” hint is “safe” in the sense that, if somehow we get interrupted between the previous instruction and the next instruction, so that the value is written back to the Register File, no harm is done. Even so, with clauses, this sort of interruption should not be possible.

This handles code like

```
ADD r3, r1, r2
SUB r4, r3(last), r5
```

With a very small additional change, we can also handle code like

```
ADD r8, r6, r7
SUB r8, r8, r9
```

Now the interesting feature is that the value `r8` is immediately overwritten. So if we add some small logic in the Copy Back Stop block to check that the destination register of the current instruction matches one of the input operands (we already have similar logic to check that we can use bypass) then we can again avoid writing back `r8`, at least the first time as a result of the `ADD`.

So we have a scheme to avoid one register read per instruction (sometimes) and one register write (sometimes).

Of course we can do even better. These bypass schemes only allow special handling of the most recent previous result. What if we provide some storage for some number of the last few results, whatever makes sense (maybe four, maybe eight)?

We could manage this in multiple ways. Mechanically, like a FIFO queue, so that results go in, stick around for say eight cycles as they move to the head of the queue, then get lost. At any stage while in the queue, they can be accessed as an operand. Or we could manage it like a cache. We could write back every result, or we could use a second hint bit, now attached to the destination register rather than an input register, saying to cache this result. The design in this patent is to use a hint bit, which controls the second Copy Back Stop 620 controller.

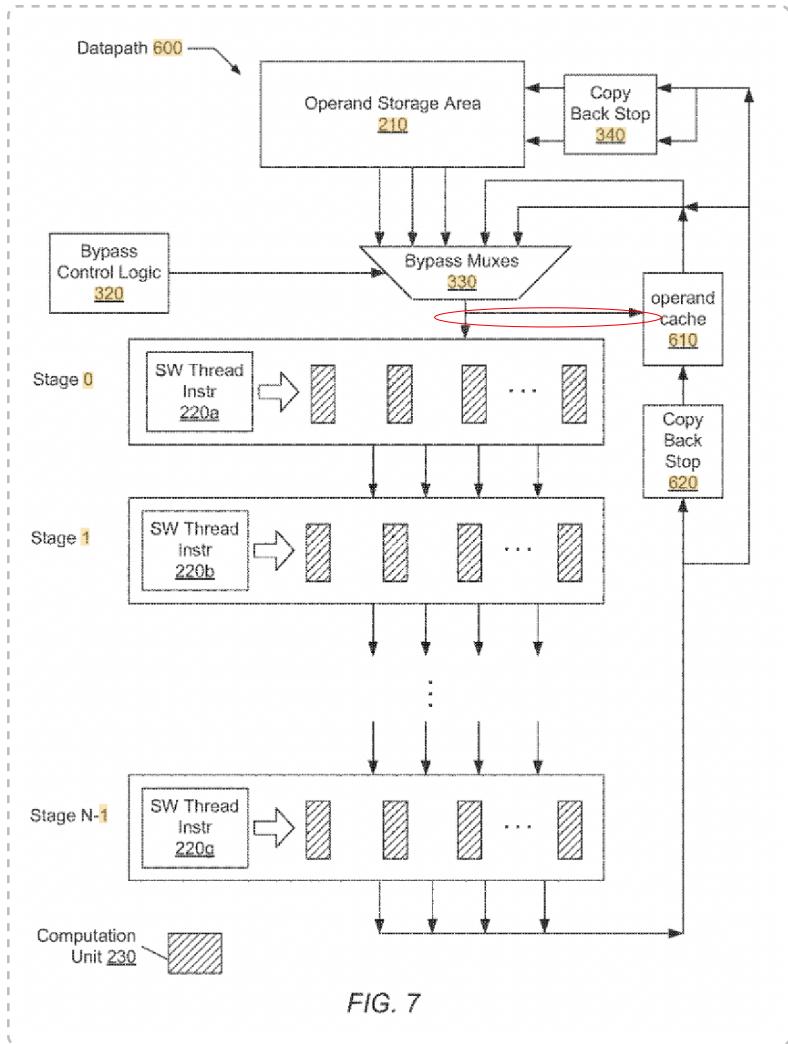


FIG. 7

Consider this new design as shown above, but for now ignore the arrow inside the red ellipse. Within this design, the “operand cache” can only store *results* of previous operands. Assuming this, what additionally can we do?

The most interesting new option is to delay the writeback of register results. Thus there will be a period during which the only valid value of a register will be the value in the Operand Cache. How does this play out?

Some operands are temporaries. After the last read of such a value (already indicated by a hint) we can mark the value in the operand cache as invalid and no writeback is required.

Other operands may eventually be used many cycles from now, and they will naturally age out of the cache, simply being written back to the Register File at some point when they are evicted.

The case that has to be treated carefully is the case where the final result in a register will be reused soon by another execution unit (eg we have calculated an address that will be used by Load). In this case the compiler needs to mark the destination register as non-cached, and the Operand Cache has to invalidate the value it holds during the process of writing the final result back to the Register File. But thinking through these cases we see that delayed writeback is indeed feasible.

The very last stage of design evolution is when we add in the wiring within the red ellipse, and so allow the Operand Cache to hold not just result registers but also input registers; along with yet another hint bit, now to every input operand, suggesting, or not, persisting that input register in the Operand Cache. Along with this final evolution, Apple also added the preloading scheme I have already discussed: a prior instruction may use an “unused” operand slot to load an operand which is routed by the bypass mux to the Operand Cache without even being picked up by any of the execution lanes.

It's interesting to me how you can evolve to the same concept of a “Register Cache” either by starting with Bypass and making it smarter (the route Apple took) or by starting with the Operand Collector and making that smarter (the route nVidia apparently took)!

There are definitely advantages to the route Apple took, in that they force you to think about writes (and synchronization/coherence) in a way that's unnecessary for a read-only (ie input operands) cache; but with the payoff of being able to handle temporaries, avoid a lot of write traffic, and capture more values in the cache.

At this point, if you're paying attention, you should notice a tension! The above description of the operand/bypass cache assumes the execution of a single thread (so that the compiler has a fairly good idea of what will be in the cache at any given time). The scheme will still work if values are forced out of the cache by new values, by preloading particular will not work well if energy is wasted preloading values that are aged out of the cache before they are even used.

I'm not sure how this tension is handled. One can imagine rigid schemes (like there are four fixed size enclaves in the Operand Cache, and at any given time four threads are allocated to these, a thread losing its slot when it moves to inactive); or flexible schemes, like each register in the cache is identified by warpID and registerID, and everyone fights for space against everyone else, may the luckiest thread win.

Certainly this design validates the point that, insofar as possible, you want thread scheduling to stay within a single warp (greedy scheduling) for as long as possible, until an instruction dependent on an earlier instruction is forced to wait a cycle or three and we switch to a second warp.

Note the date of this patent: 2013. To compare, the first Apple GPU shipped with the A11 in Sept 2017... Ultimately the details of this patent don't matter because it seems like mainly a preliminary design

(perhaps to capture the idea of caching results), not a shipped design. We move on to

optimized 2013 design

(2013) <https://patents.google.com/patent/US9378146B2> *Operand cache design.*

(2013) <https://patents.google.com/patent/US20150058571A1> *Hint values for use with an operand cache*

(2013) <https://patents.google.com/patent/US20150058572A1> *Intelligent caching for an operand cache*

The 2012 design looks pretty good. The improvements above are minor tweaks.

- What was called the Bypass Mux above, now called the Source Selection Unit, is now given a path to the Register File, so that when an operand is read from Operand Cache, as well as going to the execution unit, it can also be forwarded to the Register File.

What's the point? My guess is consider the following set of circumstances
+ we read an operand that hits in the Operand Cache,
+ the operand is NOT marked with a cache hint (so we won't be reusing it soon)
+ the operand IS marked as modified (so it will need to be stored in the Register File at some point)
In this case, we can perform one Operand Cache read to achieve both the Execution Unit operand read and the writeback to the Register File.

Another cute example of forwarding the patent suggests is the following.
We have a result path from the execution unit to the Register File, for saving a result register with no caching, bypass, or anything else.

What if we connect that path to the output of the Register File? What's the point?!

Well standard bypass is only useful for feeding the result to an immediate successor instruction with just the right timing. We can get around that storing the result in the Operand Cache, but that uses up an Operand Cache slot and takes some read/wrote energy.

If we fake that the result value just appeared on the outputs of the Register File, then it gets stored in a latch till the timing of whenever we deliver a result from the Register File to an Execution unit. So, without using either the Operand Cache or paying read/write storage costs, we get some flexibility in being able to use the result at two different clock cycles after the producer instruction, not just one particular clock cycle after the producer instruction! Very clever as a way to get a secondary use out of the Register File output latch.

This is especially valuable if the result is a temporary (ie a “last use” case) so that it never needs to be stored anywhere, not in Register File, not in Operand Cache.

A final version of this idea works for prefetched registers. Suppose that a register value has been prefetched into the operand cache cycle N, for use in cycle N+1. If we get the timing correct, we can instead execute this as having the register file deliver the result one cycle later, so that it's present on the bus going into the Execution Unit in cycle N+1 (as required for execution) and can also, in that same cycle, be forwarded into the Operand Cache. Once again we achieve the results we want, but we avoid one read of the Operand Cache!

- We add a tiny Operand Collector (maybe one instruction's worth of registers) between the Source Selection Unit and the Execution Unit. Now if an instruction blocks because eg one of the operands is not available in cache and has to be read from the Register File we can switch to a different warp (hopefully now with all its registers in Operand Cache), and switch back to the blocked instruction two or three cycles later. An easy addition (if you only collect one instruction's worth of registers) that does its small bit to improve performance.

There's a suggestion that this same Operand Collector storage can also, under different circumstances, if the Operand Collector storage is empty, be used as a temporary buffer for results.

Suppose the Operand Cache is full and the register selected to be evicted is a modified register, so it has to be written back to Register File, and a bypass result is supposed to be stored in the Operand Cache. Then the Operand Collector storage can act as a buffer for one cycle or so, rather than losing one cycle of compute while we wait for data to move around.

- It's pointed out that while prefetching to the Operand Cache is an obvious and easy way to handle prefetching, it's not necessary. At times you may want to perform a prefetch (to make a value available without delay, or to avoid bank conflicts while reading multiple registers from the register file), but you don't want to re-use the value. Again we can provide some very simple in-line storage, perhaps just one entry in size, perhaps the Operand Collector, or perhaps a simple queue of two or three entries, that can hold prefetched value(s) as an alternative to using the Operand Cache.

- There are various specific implementation details, eg to save power, that seem probably obsolete, except the suggestion of multiple operand caches (the patent suggests 4 caches each holding up to 8 registers, for a total capacity of 32 registers) for different currently active warps. Is this a better choice than a single unified cache? It's certainly simpler! It may also be more energy efficient, even if it's less area or performance efficient.

- Along with the obvious cache status bits (valid, modified, most recently used, ...) there's a "prior warp" bit that is used on context switch so that rather than wiping the entire cache (for that particular active warp) on context switch, all dirty entries are marked with this "prior warp" bit and are written back to the register file when a data-path is available.

So they are "invalid" in the sense that they should not be read by the new warp; but they are not invalid in the sense that the slot should not yet be overwritten.

This means that there is no delay in context switching, and the new warp using this cache can immediately start making use of at least some of the cache entries. (I think this mostly comes into play when a "memory miss" context switch occurs, in other words one of the active warps hits a slow instruction, like a cache miss, and is swapped out of the set of active warps.)

(2015) remove bypass and forwarding to simplify wiring

The above design sounds pretty good, but things need to evolve as process evolves. The big problem with the 2012/2013 design is that it requires a lot of wiring to handle all the bypass and forwarding

paths to handle all these extra cases where we try to avoid having to read or write data given special timing situations like using a value in the next cycle or two. And wiring is increasingly becoming more of a design constraint than the number of transistors.

So with (2015) <https://patents.google.com/patent/US9619394B2> *Operand cache flush, eviction, and clean techniques using hint information and dirty information* we take all the best ideas in the earlier design, but modify details so as to use much less wiring.

The first big change is that now every operand and every result flows through the operand cache. This removes a whole lot of wiring; no special purpose bypass/forwarding paths.

In principle flowing through the operand cache doesn't have to mean allocation in the operand cache; you could imagine the value simply using the common set of wires to flow through the cache on the way to either the Register File or the Execution Unit. But the decision appears to have been made that in fact everything that flows through the cache will be recorded in the cache – perhaps experience and simulation showed that such a high fraction of operands were cached that optimizing special cases is not worth it?

This in turn means that the semantics of the cache become somewhat different (and in fact richer). The default now is that every input and every result is cached, so a “cache retain” hint becomes a “long term retain” hint, what I called a “hold” hint in my introduction. The compiler can do something like treat say a reuse within three instructions as being the default case, and a reuse for longer than three instructions as being the “retain hint” case, so we effectively indicate low vs high priority values in the cache when it comes to eviction.

The second change is that we want to avoid a few cycles of slowdown whenever it's necessary to, more or less, flush the cache.

You will recall that in the previous design we had the Datapath with first priority access to the Register File, and everyone else as second priority through a Read Queue and a Write Queue. If we go by the diagrams in the patents, then everyone (including Datapath) now shares the Write Queue. This means that bandwidth to write to the Register File may sometimes be limited, and we don't want the Operand Cache to have to halt for a few cycles because there are no free entries and all entries are dirty and need to be written out.

To limit the chance of hitting such a situation, the Operand Cache eviction priority looks something like
1st to evict: clean, low priority

2nd to evict: dirty high priority

3rd to evict: clean high priority.

Dirty, low priority (ie a result that we do not expect to reuse soon) is, in a sense the lowest priority of all. It is immediately written to the Write Queue. If the Operand Cache “seems to have space” (metric determined in some convenient way) it will be kept in the cache as a clean but low priority entry that may be used in the future; otherwise it will be immediately invalidated once it is in the Write Cache. (There's obviously scope here for optimization in the terms of temporary buffers and suchlike...)

The implementation of this priority scheme can be achieved in various ways, for example via the

method I described of choosing a random line and if the choice is in a protected class, trying again, each time raising the protected class.

We don't immediately write out dirty high priority values because, in many cases, these are temporaries that will be overwritten. Ideally many of these temporaries eventually turn into last_use registers and do not even need to be written out.

But we may start to write out these dirty high priority values if both the cache starts to become extremely full of dirty values and the Write Queue is getting full. Evicting dirty high priority entries before high priority clean entries doesn't necessarily help in terms of performance (dirty vs clean doesn't indicate too much about when an entry might be reused) but it does smooth out the writing of values from the Operand Cache through the Write Queue to the Register File.

The third change that is described is what appears to be more fine-grained way to flush the cache. We have already seen that machinery was in place to handle when one warp stopped using the cache and another warp started, ie at the end of a Datapath clause the Operand Cache could be flushed simultaneously with a new warp beginning to use the cache.

What's added is that at least some instructions have control bits that can flush the cache.

The patent doesn't explain why this is useful, my guess is something like:

Consider situations like returning from a function call. In the function call, registers were used in various ways that are probably not relevant to when the function returns. It's probably most sensible, as the last instruction of the function, to "flush" the cache. There may be one or two dirty registers (representing the final result of the function), and various registers that are marked as high priority. It would make sense to "flush" the cache in the sense of writing out the dirty registers and either invalidating all registers or marking them all as clean but low priority.

This doesn't even need to be exactly a function call; it could be something like an inlined function, namely a block of work that uses and reuses many registers but which, afterwards, the compiler knows that the registers used will not be reused again for many cycles and so there's no need to hold onto them.

The new functionality then, is the addition of what is effectively a "cache flush" bit to at least some instructions. Still a cache control bit, but now controlling the entire cache, not just per-entry control.

2015 per-lane validity/dirty flags

What happens to the Operand Cache under predication? We have speculated that we can handle the Operand Cache via reading a to-be-written operand before predication. But this read before write is clearly not ideal in terms of energy or bandwidth. Another, also not great, alternative is something like having either the compiler or the hardware disable writing result registers to the Operand Cache while under predication.

The next two patents deal with predication.

(2015) <https://patents.google.com/patent/US9785567B2> *Operand cache control techniques*

The big idea now is to augment the overall “valid entry” bit for a register cache entry with 32 per-lane valid bits.

A single validity bit for an entry implies that all 32 lanes of the entry are valid, but they may not be if the entry was allocated under predication. These per-lane validity bits are then updated as appropriate as destination registers are written to the Operand Cache under predication, and used as appropriate when the registers are either read, or written to the Register File.

In fact what you really want is two per-lane bits, to indicate invalid, valid (but clean), and valid (but dirty).

Something like this was always a good idea, but becomes essential now that all operands flow through the register cache, not just some of them.

We also get technical details of how some fine points are handled.

Although the GPU pipeline are not out of order, they are pipelined, with operand cache control (eg allocation or de-allocation) happening in different cycles that may be separated by a few clock ticks.

This means we have to be careful with some issues. For example:

How should we handle a last-use operand? Obviously the end goal is to invalidate this cache entry (no need for writeback, no need for later reuse) right after the value is read for the last time. But how to implement that?

Suppose we have a single bit-field that means something like “pending last use” which we set for say register R2 when we perform cache control for instruction N in cycle N . Then in say cycle $N + 4$, when we read the register value we invalidate the cache entry.

The problem with this is that suppose that instruction $N - 1$ also reads register R2, in cycle $N + 3$? It will see the pending last use flag, and invalidate the entry! So we instead need something like a “last use instruction ID” which we can match each cycle against the instruction that is reading the register values to make sure of a correct match.

Another case discussed, to be handled carefully, is within-cycle timing. Once a last use register has been marked invalid, we can reuse it. In some sense the “easiest” way to do this is to have it available for use in the next cycle. But we can be more aggressive. We expect reading register values to occur at the start (let’s call it the first half) of a cycle, and results (from a previous instruction) to be available at the end (let’s call it the second half) of a cycle. So if we get the timing details right, we can route a destination into what was a last-use register in the same cycle that we read from that register.

The final case discussed is how `last_use` interacts with predication. What the patent specifies is clear, and will work, but is also sub-optimal. The rule is that `last_use` applies to the entire entry, not per-lane values. Consider the consequences.

Suppose we have a basic `if () then {A} else {B}` type structure, and within both A and B we produce some temporary registers, let’s say R3. So we want to indicate that R3 is dead by the end of this block of code.

If we mark R3 as `last_use` within the A block, then we have invalidated it for the B block. Not a catastrophe in terms of correctness (the lanes that are not written out will not be read by B) but not optimal in terms of latency and energy use because we have to re-read from the Register File.

So let’s not mark R3 as `last_use` in the A block, only mark it in the B block. That sounds good, but what

if there is no else case, all the lanes fall through case A? Then presumably the instruction hardware is smart enough to detect an empty predication mask and simply skip these instructions? So again not incorrectness, but the value is not marked as last_use, and so we waste some time slots in the Operand Cache, and then some energy writing it back to the Register File.

Neither of these are great options!

It would be nice if there were an easy non-predicated way to simply invalidate R3 after the join, past the end of the B block, something like a `NOP_LAST_USE R3 (last_use)` instruction, that does the bare minimum of wiping the register, without any extra work or producing a result register. Maybe there is such a thing, it just hasn't been found yet?

2016 interaction of per-lane flags with *speculative predication*

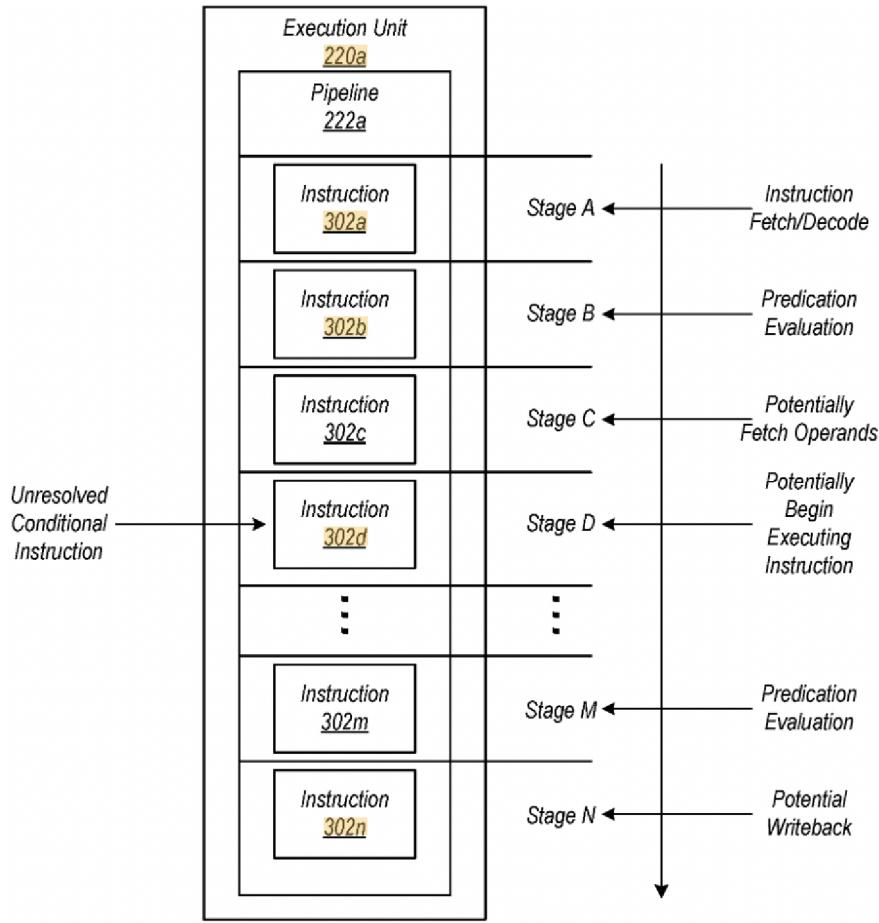
These per-lane flags sounds pretty good. And yet, one year later we see limits to how they can be used! Let's look at the patent, then we'll discuss details:

(2016) <https://patents.google.com/patent/US10613987B2> *Operand cache coherence for SIMD processor supporting predication.*

The headline feature is extremely obvious: the 2013 GPU predication patent that we already discussed essentially says

- “if we don’t know the predication mask at the start of execution, compute all lanes, then mask off which lanes we write to the *Register File*”; this new patent says
- “if we don’t know the predication mask at the start of execution, compute all lanes, then mask off which lanes we write to the *Operand Cache*”.

You can see the idea here: [here](#)



The other thing it points out (probably not especially usable or useful) is that if we perform this speculation, then a side effect of *operand fetch* will occur which, in theory, might not have happened if we knew the predication mask was all false and so skipped the instruction.

More important is the grand table of every possible situation:

<i>Destination operand miss in operand cache</i>	<i>No unresolved conditionals in pipeline</i>	<i>Instance of instruction is known-to-execute</i>		<i>Mark operand cache entry state as dirty and do not read register file</i>
		<i>Instance of instruction is known-not-to-execute</i>		<i>Mark operand cache entry state as undefined and do not read register file</i>
	<i>At least one unresolved conditional in pipeline</i>	<i>Instance of instruction is unresolved</i>		<i>Mark operand cache entry state as dirty and read register file</i>
		<i>Instance of instruction is known-to-execute</i>		<i>Mark operand cache entry state as dirty and read register file</i>
		<i>Instance of instruction is known-not-to-execute</i>		<i>Mark operand cache entry state as clean and read register file</i>
<i>Destination operand hit in operand cache</i>	<i>No unresolved conditionals in pipeline</i>	<i>Instance of instruction is known-to-execute</i>		<i>Mark operand cache entry state as dirty and do not read register file</i>
		<i>Instance of instruction is known-not-to-execute</i>		<i>Do not change operand cache entry state and do not read register file</i>
	<i>At least one unresolved conditional in pipeline</i>	<i>Instance of instruction is unresolved</i>	<i>Entry is dirty</i>	<i>Leave operand cache entry state as dirty and do not read register file</i>
		<i>Instance of instruction is unresolved</i>	<i>Entry is clean</i>	<i>Mark operand cache entry state as dirty and do not read register file</i>
		<i>Instance of instruction is unresolved</i>	<i>Entry is undefined</i>	<i>Mark operand cache entry state as dirty and read register file</i>
		<i>Instance of instruction is known-to-execute</i>		<i>Mark operand cache entry state as dirty and do not read register file</i>
		<i>Instance of instruction is known-not-to-execute</i>		<i>Do not change operand cache entry state and do not read register file</i>

Note the cases where we force a read from the Register File.

Why can we not mark the predicated destination lanes as per-lane valid and dirty, and the non-predicated destination lanes as per-lane invalid? It seems like the patent is ignoring the entire point of the previous patent! But it's the same authors, so that's clearly not the case!

The next patent explains the issue to some extent.

(2018) aggressively pipelined GPU / “virtual operand cache allocation”

We have a situation somewhat analogous to the earlier bypass/forwarding situation: we think of some wonderfully clever ways to optimize one particular design, then year later think of an even better restructured design, for which the earlier optimizations no longer make sense!

The main restructuring here is of the GPU pipeline. We've kind of assumed a very obvious, slow, and not especially pipelined instruction flow where, conceptually at least, in one cycle we do everything related to the Operand Cache: allocate an entry or two, start Register File Read (if necessary), wait until Register File Read is done, move on to Execute.

This has been substantially pipelined in a way that we will describe soon, but for now, consider some consequences.

In particular assume something like `if () then {A} else {B}` and assume the last instruction of the A block, followed by the first instruction of the B block, look something like

predicate A: instruction *writing* partially to R3

predicate B: instruction *reading* partially from R3

If we have pipelining, now we have a situation where say in cycle N we check the operand cache to be sure that the values needed for predicate B are present. Meanwhile later maybe in cycle $N + 7$ we write out a different set of lanes values to register R3. And we assume that we're not even sure, in cycles $N - 1$ and $N - 2$, when we started executing these two instructions, what the predicate mask looks like!

It starts to turn into an awful mess!

In theory, with enough additional per-lane predicate bits, we might be able to construct some sort of super-optimized finite state machine that can handle every case optimally. But if the most common case is going to be something like

- the set of A predicated instructions is short (two or three say), meaning that
- by the time we start the operand-fetch of the B predicated instructions we still don't even know yet which lanes A is going to be writing out

then our options are somewhat limited! For example we could start renaming registers, so that the the A result R3 is different (and occupies a different Operand cache slot) from the input of B R3; but that seems like overkill!

The more you think about it, the more you conclude that the overall best option is, at least for these speculative cases, going back to loading the entire register so that what's in the operand cache is always consistent across the whole register.

You'll get a better feeling for why this pretty much has to be so when we consider the pipeline over the next few patents.

So, forget the previous complications with predication and just consider a basic pipelined GPU. The details are not yet clear, but the pipeline is going to have to look something like

- check that the source and destination operands are in the Operand Cache. If not, allocate as many new entries as required.
- load the source Operands and wait for them to arrive
- execute the instruction
- write out the result

Now compare this to the Allocate (usually called Rename) stage in a CPU. The GPU suffers from the same problem as the CPU:

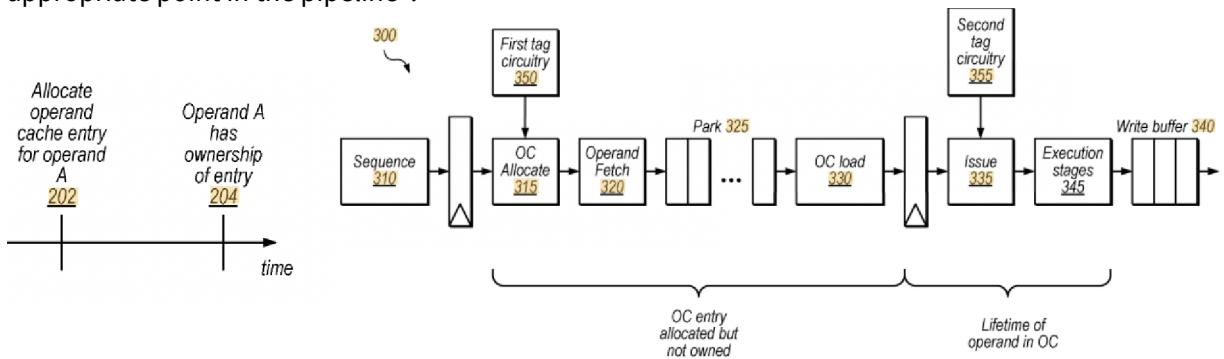
we are forced to *Allocate* a resource early in a pipeline, but we only need the physical resource (physical register storage or LSQ storage for a CPU, a slot in the Operand Cache for a GPU) much later in the pipeline. Being forced to allocate the *physical resource* early leads to wasted resources.

The solution in each case is the same, though the details differ: perform a *virtual allocation* that handles all the control complication that is required, but defer the *physical allocation* to as late as possible.

We see this described in (2018) <https://patents.google.com/patent/US20200065104A1> *Pipelined Allocation for Operand Cache*.

In the case of the GPU the way this plays out is that

- the control complication (leading to virtual allocation) is deciding which entry to evict to make way for the new entry
- but rather than evict immediately, let's leave the entry in place for as long as possible (which could 10 or more cycles, if we have performed allocation for a destination register); after all a successor instruction may still want to read the value we are about to evict
- so we have kinda split register ownership into two parts. There is *physical ownership* which specifies that this slot corresponds to register R3 and that it is modified, and perhaps that it is a last use register invalid after instruction N; and there is *allocation ownership*, which means something like “instruction M is going to take possession of this (for source, or destination operand) once instruction M reaches the appropriate point in the pipeline”.



The diagrams make this clear: At OC Allocate we decide to give slot K to the first operand of the instruc-

tion, but we don't invalidate what is in slot K right up until the end of Operand Load which, I think, means "write the result loaded from the Register File into the Operand Cache".

How do we implement this? Essentially by giving each Operand Cache slot two tags, one which represent ownership (meaning something like "is caching register R3 of warp W") while the other represents allocation (so meaning something like "*will* cache register R4 of warp Y").

This gives the big picture of the new pipelining structure. Let's now consider a few details which are a consequence of operations that used to occur in one cycle being split over multiple cycles.

- There are now 4 per-lane states; a lane of a register can be invalid, valid (but clean), valid (but dirty), or *Pending valid*. Pending valid means that the lane might be changed by a speculative predicated instruction, or might not, to be resolved at the end of execution when we decide which lanes to write.

- I suggested that in a perfect world register ownership for a destination register would only change at the point of writing the instruction result. However in this first implementation we go with a simpler scheme whereby the timing is

- + the register slot (for read or write) is allocated at OC Allocate
- + ownership changes (again for read or write) essentially at Issue

- this is sub-optimal, giving dead cycles between Issue and the actual write to the Operand Cache. To try to limit this, each Operand Cache Slot also has a "Cycles Left" field which, if I understand this correctly, counts how long the slot must be held (ie cannot be evicted) until its destination result will be written. This is used, among other things, in Allocation – if we are forced to choose a Slot that require writeback of a modified result, we will have to wait (in the sequence of Park stages) until that slot has been moved to the Write Queue. Even worse, if we are forced to choose a Slot with a non-zero cycles_left value, we will have to wait in Park until the controlling instruction has finished and written out the value, and then that value has been moved to the Write Queue.

The pipeline (in this version) is still one instruction wide, completely in order, in the sense that instructions enter the pipeline from Sequence (which you can think of as some appropriate scheduling of a next instruction from the clause of one of the active warps), and then flow down the pipeline without reordering. At Operand Fetch we test if any operands are in the Operand Cache and if not we load them from the Register File. We then wait for some variable number of cycles in Park until

- all required operand slots become free (which may mean waiting for transfer out of dirty registers, or even first writeback of earlier destination registers)
- all required operands have been loaded from Register File

after which we can move to Issue and real execution.

This is not obviously an improvement on what we had before! There's much the same level of in-order waiting around for something to happen, with more complexity to handle the split timing. So what's the point?

Two points: The first is that we can run at higher frequencies. The second is we're laying the groundwork for, one step at a time, getting rid of all this waiting around doing nothing...

As already pointed out, this patent again reminds us that we now have a single unified Write Buffer sitting between the Operand Cache and the Register File. You will recall that we began with direct

Datapath access to the Register File and a Write Buffer for all the other the secondary execution units. This is yet another example of redoing ideas that, at the time, made sense. We have reverted to a single Write Queue (and, as we will see, also a single Read Queue).

- To some extent this is because the more the Operand Cache captures most Datapath register file traffic, the less need there is for a higher priority mechanisms to handle access to the Register File,
- to some extent it's because these centralized Read and Write Queues have become more sophisticated (as we will see),
- and to some extent these centralized Read and Write queues allow for a new way to allocate the Register File. Skipping forward in time, we earlier discussed how the A17/M3 “virtualize” all memory resource allocation including, specifically, Register Allocation. This means that all accesses to the Register File need to go through some sort of TLB to map the (register+warpid) to a physical SRAM address. Even in these earlier designs of 2017 or 2018 we may still have implemented some sort of reasonably sophisticated remapping scheme to allow for maximal packing of registers across different warps, and this was probably simplified by having a single access point into the Register File, by a single per bank Read Queue and a single per bank Write Queue, both going through a single Register Remapper.

Regardless, while giving up one idea from the past, the patent revives another idea, namely bypass and forwarding!

Suppose that a modified register has been evicted from the Operand Cache. After looking (unsuccessfully) in the Operand Cache, the next place to look is the Write Queue (starting at the entry, since any value there is the newest value, and newer than anything in the Register File). If we find a matching entry in the Write Queue, then one option would be to simply pause until the value is written to Register File, and then read it; but obviously we'd prefer to forward it back to the Operand Cache.

Along with this patent exemplifying one of our common principles (virtual vs physical allocation) we also see the start of evolution down an unexpected path. We're beginning to view the problem of Operand Cache optimization as something like CPU Resource Allocation, using ideas from register Renaming. Where will this take us...?

We're gradually evolving our way from our starting point of the simplest possible in-order barrel processor to something like an SMT+SoEMT design which is almost ready for a small amount of OoO execution (something perhaps like the PPC 601 and machines of that generation)...

(2020) multiple frontends to common execution backend

Consider the Datapath pipeline from a structural viewpoint.

What we would like is that every cycle a new instruction is issued. What prevents that?

A first limiter is dependencies between instructions; an instruction that uses an input the result of an earlier instruction cannot execute until that earlier result is available. We try to deal with that by

- scheduling by the compiler to interleave independent instructions between dependent instructions

- having as large a pool as feasible of active warps, and scheduling from different warps until the result required by a particular warp is available.

We can also speculate (to a very limited extent) through unknown predicates.

A second dependency results from the Operand Cache. We need the operand cache to deal with the energy and bandwidth costs of the very large register file, but there are uncertainties involved with the Operand Cache. We can try to arrange things so that values are available in the Cache, but we can never be sure. And remember that we are an in-order pipeline.

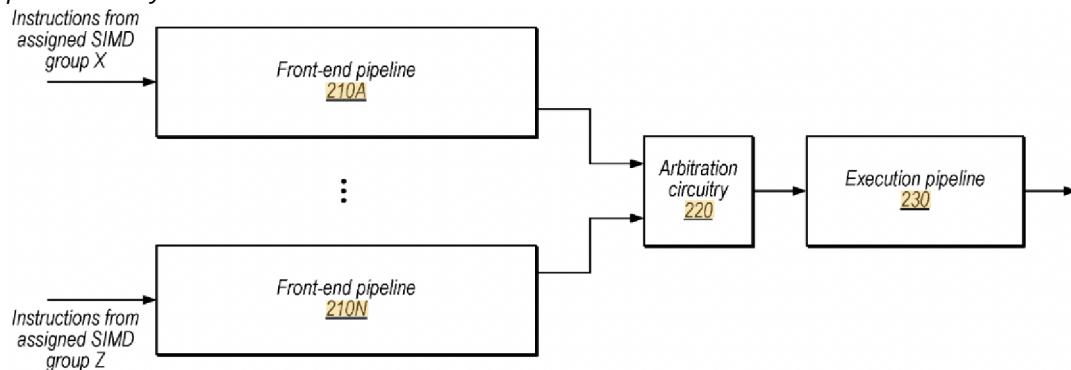
This means that at the point where we check the Operand Cache for the presence of operands, we have to pause if those operands are not present, and being in-order specifically *means* that this is a pause; we can't let a later instruction that does have all its operands begin execution. We similarly have to be pessimistic about instruction scheduling in that we must assume an instruction generating a value we will later use may take the longest possible amount of time (waiting for all operands, and writeback of all required cache slots) to generate a result.

A third occasional limiter is when we issue instructions to the Special Function Unit, which can only accept a new instruction every few cycles, not every cycle, so it will saturate if we give it too many instructions back-to-back.

Is there a way to work around these occasional lost cycles without going full OoO? Yes there is, because we are executing SMT.

Let's start by assuming multiple "mini-pipelines" that exist up to the point of Issue. Thus a given mini-pipeline can stall, waiting for operand collection, while another mini-pipeline moves forward.

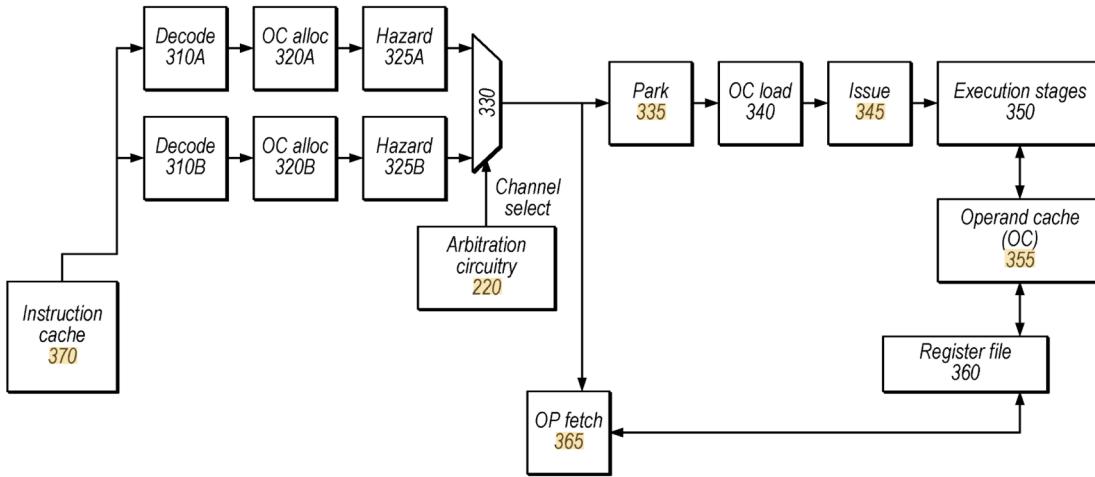
We can see this design in (2020) <https://patents.google.com/patent/US11422822B2> *Multi-channel data path circuitry*.



So assume we have two such pipelines. Then more or less each of our active warps (which means essentially a clause giving the instructions, plus a register set) is assigned to one of the two pipelines. This might be something static like even vs odd warpID, or something more dynamic; point is that if we

have say six active warps, then three warps will continue to be multiplexed into the front-end pipeline, but if this is forced to halt, the other pipeline can keep going, switching through its three warps.

In slightly more detail we see this below:



Given what we already know, presumably

- Instruction cache is an L0 clause cache.
- “Decode” is fairly minimal decoding; it appears to mostly mean cracking of complex instructions into a sequence of simpler instructions.

If there are any sorts of latencies associated with Instruction Fetch (maybe two or three cycles to move a block of 4 or 8 instructions to a local buffer?) or with Decode (maybe a cycle to switch to executing a microcode engine and then to switch back?) then one pipeline can take the delay without hurting the other.

- OC Alloc involves checking for the presence of operands and, if necessary, allocating cache slots. It also involves rewriting register dependencies so that the instruction changes from say *reading register R1, R2 to produce R3* into *reading slots S7, S8 to produce an output destined for slot S9*.

- Hazards are still handled by a stall counter, ensuring that an instruction is not released until at least n cycles.

I think the explicit Hazard stage is something to do with collecting the data required to establish this count. You'll recall that earlier we mentioned each destination Operand Cache entry has a "cycles till done" value which was used to decide when allocating Operand Slots, to try to avoid slots that would have to delay waiting for a result. We can use those same values, just look them up for the source operands, to decide how long we need to wait before it's safe to release the instruction.

- You might wonder why there is a GPU issue stage. Surely the dependency checks normally performed by Issue were handled earlier at Hazard/Channel Select?

The idea seems to be that, for performance reasons, Channel Select can't be *exactly accurate* in the instruction it chooses. It's trying to balance multiple factors to ensure a flow of instructions without bubbles, and using what may in some cases be guesses as to the latencies of when values will be present in the Operand Cache or when a Complex Unit will become free (this is *somewhat* like Speculative Scheduling in an OoO CPU).

So we have a second stage of Issue that checks that everything is in fact completely correct with the (hopefully rarely exercised) option to insert a bubble if necessary, if a dependency has not been satisfied or the target unit is still busy.

I don't fully understand the timing model. It makes sense, and appears to be the case, that delays waiting for Operands to be *calculated* are handled by a variable delay in the Hazard stage. You'd think that's also a good place to wait for operands to be *fetched* from the Register File but you'll see that Operand Fetch (hopefully a rare operation) is placed on the other side of the mux, and we still have to wait for operand fetch (if that was necessary) in the Park stage.

This seems sub-optimal, but obviously it's somewhat tricky to balance the timing into the Operand Cache of multiple instructions all trying to fetch or prefetch registers and to write registers, along with the machinery we discussed earlier of trying to make physical register allocation as short as possible.

You might want to think about it. Consider all the elements required to try to ensure that fetched registers and calculated registers have a slot available for writing, and are definitely available in the cache for reading, possibly by multiple different instruction, all the while subject also to eviction and early allocation!

Maybe this is a first iteration, and a year or two later we will get a design with Operand Fetch moved to the other side of the Channel Select mux?

It's interesting to see how this is converging a variant of anOoO design, just with different names.

The OC (Operand Cache) is more or less the equivalent of the CPU Register File, and the GPU Register File is (kinda sorta) like a CPU L2 Register File, as used by eg IBM POWER.

OC Alloc is like CPU Rename.

The stages Decode, OC Alloc, and Hazard are each one instruction wide but "four instructions deep", so they are somewhat like a 4-wide version of a CPU, only without any dependency checks required

between instructions because the four active threads are all independent.

The Hazard stage is somewhat like instructions sitting in a scheduling queue waiting to become runnable (ie all dependencies satisfied) and the Channel Select Mux is like CPU issue. Park and OC Load are like the Register Load stages of the CPU.

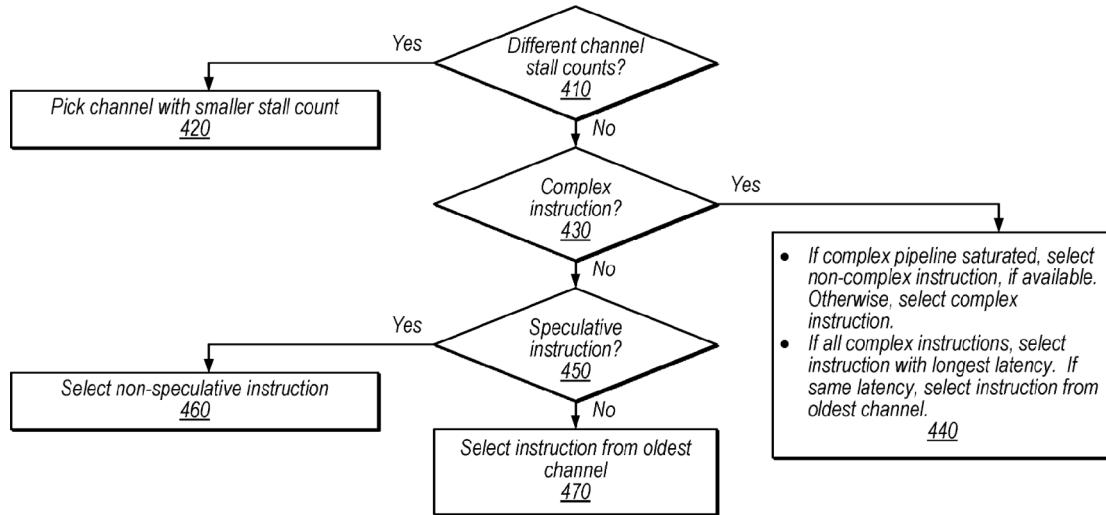
The other obvious aspect of this design is that it seems fairly easily scalable.

For example you could move up to three front-ends, making it more likely every cycle that we can find an instruction ready to execute.

Alternatively, recall that there are multiple different pieces of hardware that live in Execution, including an integer ALU, a wide FP unit, a narrow FP unit, and a special purpose functions unit. If we doubled the Park/OC Load/Issue pipeline, and allowed Channel Select to pass two instructions per cycle instead of one, we would have a superscalar GPU, something like say the Pentium with its U and V pipes; this could dual issue various combinations of instructions targeting these different Execution Units.

And of course we could combine these; three front-end pipelines, feeding two middle pipeline, dual issuing to two of four or so execution units.

Here's what the Channel Select logic looks like, approximately:



In this flow chart, by “channel” is meant warp, and the “stall count” is, approximately, an estimation of, if the instruction is released into the next stage of the pipeline, how long it may be delayed, either waiting for an operand or for an execution unit. So we can release instructions that may still be waiting for an input being calculated; presumably they will wait in either Park or Issue until the input has been calculated.

One minor advantage (perhaps) of releasing these not-yet runnable instructions is that, even if their (not-being calculated) operands are already present in the Operand Cache, if they are prefetching registers for a later instruction, the prefetch begins right away?

Another way to do this might be something like cracking the prefetch part of an instruction into a separate piece that can move on its own trajectory?

The second aspect of the scheduling is that Complex Instructions (ie things like reciprocal or transcendental) may be only half (16 execution units) or a quarter (8 execution units) wide, so an instruction sent there will keep the execution unit occupied, even with pipelining, for multiple cycles. So we want to try to interleave these instructions with simple instructions.

Then we have our old friend, *speculative* execution, as usual meaning predicate based speculation, and as before if we can avoid speculation we do so (chance of saving energy on some lanes) but it's not a high priority compared to the other factors.

How else might we improve the above design?

The first serious operand cache in the 2013 patents was described as having something 8 entries

for 4 active threads. That 4 active threads has probably become something like 6 active threads, along with 2 or 3 separate mini-front-ends into the pipeline. But how should we organize the cache?

It's clearly easiest, and reasonably cheap, to do something like have 4 or 6 separate fully associative caches, each holding say 8 entries, a separate operand cache for each active warp; but this seems suboptimal for area, energy, and performance.

In particular there's no way to exploit free entries across these separate small associative caches, for example if one thread is making better "reuse" of cache entries than another, or if the scheduler has only been able to find say 4 active threads. Can we do better?

Consider something like the following:

Assume say 8 banks (essentially sets, but physically independent if that's valuable) each with six entries, or alternatively 16 banks of 3 entries.

We hash the combination of threadID and registerID into a bank index (either 3 or 4 bits) then look at the appropriate bank in the 3 or 6 ways to find the entry we want.

We can use credits to force each active thread to have, say, at least six entries always available, with whatever excess entries remain once all the active thread credits are counted up being available for use by all threads, essentially each thread competing against the other threads, maybe the most useful operand entries win.

We could even use the most sophisticated way predictor I described for the L1D (the one that used a hash of virtual address+ ASID into a table sized to twice the number of lines in the cache) to mostly reduce the energy costs of performing the 3 or 6 tag comparisons for entry in a bank.

The end goal of all this is that, perhaps we can, in the same energy and cycle time (though with more area) give each thread something more like 16 or 20 operand cache entries, even more so if not all 6 active threads are fully populated. There may now be some unevenness in the access times of the operand cache based on the occasional hash or bank collision, or way misprediction, but if the numbers work out, then hopefully such unevenness is rare and can be handled by the occasional rare stall at the Issue stage.

With larger Operand Caches we can strive for more cached registers per warp (if that is valuable) or more active warps (if that is valuable). Unfortunately I haven't yet encountered any papers that usefully indicate what optimal numbers for these two (number of cached registers, number of warps) might look like.

(2019) Register File QoS via catchup bandwidth

Even with the operand cache, we still frequently need access the register file, both by Datapath as a primary client and by all the other secondary clients.

(2019) <https://patents.google.com/patent/US11080055B2> *Register file arbitration* describes how that is made more efficient.

The obvious part is that we have multiple clients for the register file, and the register file is implemented as multiple banks of SRAM. These multiple clients may sometimes want to access the same

SRAM bank in the same cycle, so we need an arbitrator to accept all the requests, sort them by target bank, and decide which one request for each bank will be honored.

The problem then is the old one of how do we balance fairness and throughput?

For example, even if we give requests a priority (eg based on the priority of the kernel, which is more or less inherited from the priority of the application) we still have to handle situations like: multiple equal-priority requesters; or do we allow starvation of lower priority clients? In other words, QoS. The solution chosen every time we encounter QoS can be different because either the costs are different or the request patterns are different.

The particular solution chosen for the GPU includes the following pieces.

- We have a FIFO queue for each client, which holds incoming requests, and these queues all feed into an arbiter.

This initial FIFO decouples the register request from execution so that execution by the particular client may continue for a few more cycles (possibly inserting a more register requests into the FIFO) even if the register read is delayed by a cycle or two.

- What happens if a particular queue has its request denied by the arbiter?

One obvious possibility is to try again and keep trying, maybe boosting priority each time the request is denied.

The problem with this scheme is that it's unfair in terms of *bandwidth*, but bandwidth is what's most limited in a GPU. It's unfair in that every cycle the unit only gets to make *one* register request, even if it has frequently lost out in its requests. This is obvious when you see it, but not obvious if you think about QoS from a CPU latency perspective!

- So we fix that by creating a supplemental queue. After the request is denied by the arbiter, it moves to the supplemental queue, and next cycle both the supplemental queue and the "main" queue get to make a request to the arbiter. In the usual case that they are accessing different banks, possibly both requests might go through. In this way, the unit is given some "catchup" bandwidth.

To repeat this: each of clients A..N have their own main queue with a chance at the Register File.

Each also have a supplementary queue with a chance at the Register File.

The arbiter will make its choices (for each bank) based on whatever primary decision criteria it uses, but a client A that has requests at the head of both its main and its supplemental queues now has a larger chance at being chosen by the arbiter each cycle.

This doesn't exactly mean it has twice as high a chance of reading from a particular bank; once again that is latency thinking. Instead it means that if this client was hoping to read say two values over four cycles, in the earlier cycle it read no values, but this cycle maybe it will be able to read two values, one from each of two banks.

- Various tweaks to this scheme are possible.

- + One obvious choice is to boost the priority of the supplemental queue.

- + Less obvious is to share the queue between two clients. The pairs of clients are chosen so that they usually are not simultaneously busy, meaning that most of the time we get the benefit of the supple-

mental queue while only paying half the cost.

Even if both clients are busy, the scheme still works, it just means the storage space of the supplemental queue has to be shared.

Like some other patents we have seen, a quick hack not a general scalable solution to all queueing situations, but nice in how simple it is while able to smooth the most common overflow situations.

Instruction format

At this point we know enough that we can understand the instruction format, building on the work of Dougall at <https://dougallj.github.io/applegpu/docs.html>

Dougall did all the real work, I'm just analyzing his results, trying to clarify some things I think are not optimally explained, and making wild assumptions about how the system works (based on what would make sense given the rest of the GPU) in a way that Dougall was not willing to do!

Instructions are variable length in multiples of two bytes. The basic pattern is (more or less) 7 lowest bits specify a basic opcode, next 8 bits specify a basic destination, then 15th bit specifies whether we have the long or short version of the instruction. Thus, from these first two bytes we know the underlying length (from the opcode as 4, or or whatever bytes) and whether it does or does not have an extension of say two more bytes.

Note that it appears that the instruction set encoding and some details have changed as of the M3, but no-one has done any deep work on this so far.

datapath instructions

So as an example look at
mov (Move 16-bit Immediate)

We see the basic opcode, the “basic” destination specifier (given by D and DT, to be explained) and the L bit saying whether the extra last two bytes are or are not present.

- We have 256 16b registers, which can also be treated as 128 32b registers or even, for some purposes like memory addressing, 64 64b registers.
 - Along with these we have 512 uniform registers (ie “scalar” registers that are not per-lane) which can likewise be treated as 32b or 64b.
 - Instruction operands can come from three sources: immediates, registers (ie per-lane registers), and uniforms.

Thus an operand specifier is slightly more complex than just a register number.

Apart from specifying the length of an operand, we also specify a cache control bits or two for each operand. Remember that we have seen

- a reuse bit (for any register, source or destination) - retain in the Operand Cache
- a last_use bit (only for sources) - flush from the Operand Cache and, if modified, don't bother writing to Register File

Let's start with the simplest case of specifying a destination. In that case the only targets are normal "per lane" registers.

Bit 7 in the instruction specifies the *reuse* cache hint, meaning that the value may be used again.

Bit 8 is a Register Length specifier.

Think about a register's number, specifically the low two bits. So think specifically of registers s0, s1, s2, s3. We can also treat these as 32b registers (r0 and r2) or a 64b register (l0) (just making up names short, register, and long to give the letters s, r, and l). How can we specify the registerID and the length of it that we wish to use in the most convenient way?

consider the lowest two bits and one extra bit [a][b][c].

This gives 8 possible combinations, and think about them all.

Suppose we set [c] to 0, meaning 16bit. Then [a] and [b] are parts of the full register specifier needed for a 16b register.

Now set [b][c] to [0][1], meaning 32 bit. Now [a] is part of the full register specification needed for a 32b register; we know the register specifier (r0, r2, etc) must be even, so we don't need a value for [b].

Now set [b][c] to [1][1] meaning 64b. Now we also don't need the value of [a], we know that the 64 register specifier (l0, l4, etc) must be a multiple of 4.

Very cool!

(People who like this sort of thing will notice that one possibility remains unused; we know that [a] is unused in the 64 case, so we could do something like set [a][b][c]=011 means a normal 64b register reference and [a][b][c]=111 means something else. In principle you could grow this to things like 128b and larger register references; and maybe for some purposes that might even become valuable. Alternatively you could define that 111 case as a different register target space, though no obvious good candidate suggests itself yet.)

The set of operations that work on 64b is unclear. Dougall's more formal specification for a destination register suggests that 64b ops only work for integer add and the add part of integer multiply-add, but I don't know if that's just because he didn't try other cases (like say 64b bit-reverse or popcount)... It makes sense that, for now, 64b is considered essentially *only* for addressing, and is only supported to that extent. But there is an obvious path to extend this to many more instructions if necessary.

Given what I said, compare this now to the instruction we see above. We see that that the "length" bit, bit 8, is zero because for a 16 bit immediate the obvious target is (only) a 16 bit destination register. But note that only 6 bits are provided for D, allowing for only 64 registers – we said there were 256 16b registers! The additional 2 bits are specified in DX in the two-byte extension to the instruction.

The idea is that most shaders don't need the full 256 registers (remember we can pack more warps onto a core if the warps use fewer registers, so we can pack as many as use up our total pool of physical registers). If a shader in fact uses 64 or fewer registers, then they can be identified with just 6 bits and

we don't need the DX bits. If we also don't need any other feature of the instruction extension, we can set the L bit to 0 and save those two bytes in the instruction stream.

This technique of packing the high two bits of a register specifier in the instruction extension also applies to source operands.

This business of an optional length extension is usually the correct way to interpret instruction layout, but there are a few cases where some detail or other somewhat forces the issue and the L bit is used as an opcode modifier rather than as indication the optional presence of an instruction extension.

Let's move on to a slightly more complex example:

mov (Move 32-bit Immediate)

47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
imm32
L D 1 1 1 0 0 0 1 0 Dt
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 ? ? Dx ? ? ? ? ? ? ? ? ? ? ? ? ? ?

Note that the basic opcode 1100010 is the same, but the destination length bit, bit 8, is set to 1. This tells us it's an immediate32 and everything else flows from that, including that we must interpret the register specifier as a 32b register.

We see something similar with

get_sr (Move From Special Register)

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
? ? Dx SRx ? ? ? ? SR 0 D Dt 1 1 1 0 0 1 0
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?

But here things can't be quite so elegant :-(

It's obvious that we want to (and do) put the high-bit register specifiers (now for both the destination register and the source register) in the extension byte from bits 24 to 31. But since we want to track instructions by two bytes rather than by single bytes, we can usefully treat this instruction as a base instruction of 24 bits long with a possible one byte extension (so using the L bit in bit 15, and occasionally saving a byte in instruction length). Oh well.

Now let's look at something with a source register. All the single operand instructions have additional complications, so let's go with about the simplest two operand instruction:

asr (Arithmetic Shift Right)

39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Bt B 0 1 At A 1 D Dt 0 1 0 1 1 1 0
63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
? ? Dx Ax Bx ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?

(Basically shift each lane of A by the corresponding lane of B and store the result in that lane of D).

By now the destination specification (bit 7 = reuse hint, bit 8 = 16 bit vs 32 bit register) is familiar. But the source register is now described not just by Ax and A (to be used as you would expect) but also by four bits designated as At.

These really form two clusters.

Bits 25 and 24 (the first two bits of At) specify the “type” of register being sourced from.

If these bits are 00 then we treat the register “specifier”, ie the 8bit bit pattern constructed from A (and possibly Ax if present), as an immediate. (I think always an unsigned 8 bit integer.)

If these bits are 01 then we treat the register specifier ie the 8bit bit pattern constructed from A (and possibly Ax if present), as a *uniform* register specifier.

How do we know if we should treat this register specifier as 16b or as 32b? That's set by bit 23.

Finally there are in fact 512 (not 256) 16b uniform registers! So we need an extra bit if we want to specify a 16b uniform register. That extra bit comes from bit 22.

You will notice that one consequence of this is that uniforms are not operand cached...

(Or, more precisely, they may be cached, but we cannot control that caching.)

Finally if bit 25 is 1, then the source is the standard per-lane registers. Bit 24 now acts as a length control bit (the equivalent of bit [c], as we saw earlier for the destination register).

For source per-lane registers we get cache control in the form of a reuse bit (bit 23) and a last_use bit (bit 22).

Once again we see an unused combination, in the bits 23 and 22 are not used when we broadcast an immediate. This seems like a shame! We could perhaps use these bits to modify the broadcast immediate in a few ways, depending on what profiling shows is most commonly used. Options include

- simply giving the immediate an extra bit or two of range
- treating the immediate as signed vs unsigned
- shifting the immediate left by some amount

So we see that, for example, in this particular case, we could specify a constant value, say 1, as input A, to be shifted by a variable per-lane amount (input B). Or we could specify a per-lane value in A to be shifted by a constant (but unknown) value held in a uniform register, in B.

We also see the pattern, common in instruction sets, of a primary opcode (in this case 0101110) and a secondary opcode, in this case the 01 of bits 27 and 26. In fact this primary opcode means something like “manipulate bitfield”).

There are enough bitfield manipulation instructions that 4 variants are not enough and we use the L bit as a variant. So the three variant bit indicators are bits 27, 26, and 15

Specific versions include

000: Bitfield Insert/Shift Left

001: Bitfield Extract and Insert Low/Shift Right

001: Extract From Register Pair

etc through the expected variants, including our

011: Arithmetic Shift Right

Now compare this

bfi (Bitfield Insert/Shift Left)

39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0			
m1	Bt	B	0 0	At	A	0	D
						Dt	0 1 0 1 1 1 0

63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40
m3 ?	Dx Ax Bx Cx ? ? m2	Ct C

We now see a few more elements fall into place.

Just to assuage your curiosity, input registers A, B, and C are defined as before, along with a mask defined by the five m3:m2:m1 bits (in other words a mask of anywhere from to 32 bits long). This many bits from B, shifted by C are inserted into A. So a fairly standard bitfield insertion operation, with the options for the source, destination, and (more unusually) the location of the inserted bits to be variable, but the length of the inserted bitfield is fixed.

Compare this instruction to ASR above. The register layout is now familiar, as is the basic opcode and the three sub-operation opcodes at bits 27, 26, and 15. What we also see is that the ASR operation has (conceptually) a slot for a third register C. If we assume the obvious conclusion from the patents, then if we fill in the appropriate values for register C in an ASR operation, then we'll get a register prefetch.

What if we don't want a register prefetch? I assume if we fill in all 0s, (which means as a register, we want to broadcast an immediate of 0) this will be treated as no prefetch. Presumably in fact any broadcast value as the C register will be treated as not a prefetch.

What about filling in a uniform register specifier? Who knows? Maybe in the current design nothing happens? Maybe there is a way to prefetch a uniform and hold it briefly, for the next instruction or so, in an operand collector, without actually placing it in the Operand Cache? Maybe it goes into the operand cache, and just ages out whenever it ages out, subject to normal LRU rules?

iadd (Integer Add or Subtract)

39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0			
s1 Bs	Bt	B	N As	At	A	0	D
						Dt	S 0 0 1 1 1 0

63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40
? ? ? ? ? ? ? ? ? ? s2	? ? ? ? ? ? ? ? ? ? Dx	Ax Bx

With arithmetic operations we start to see some of the extra bits that allow a single operation to do many things (somewhat ARM-like).

The new elements above include

- the N bit means negate B. In other words calculate A-B rather than A+B.

A similar bit converts AxB+C into AxB-C.

- the S bit (more or less) means to saturate the result rather than wrapping at 16 or 32 bits. You can think of it as a modifier attached to the destination register specifier.

- the As, and Bs bits (seems redundant to have both of them) means to treat the operands as signed rather than unsigned

- the s1:s2 bits define an amount of left shift (ie scaling) to be applied to B before adding

Compare this with

imadd (Integer Multiply-Add or Subtract)

39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0			
s1 Bs	Bt	B	N As	At	A	0	D
63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	Dt S	0 1 1 1 1 0			

?	Dx	Ax	Bx	Cx	s2	?	Cs	Ct	C
---	----	----	----	----	----	---	----	----	---

Obviously same basic ideas; now the N bit and the shift apply to C, ie the part that is added.

What I find interesting in the above is how the C operand is encoded. Wouldn't it make more sense, for iadd, to have Dx, Ax, and Bx in the last byte, so that an optional prefetch "C operand" could be encoded just like imadd? Also the placement of the two s2 bits is slightly shifted relative to the two m2 bits used in the bitfield instructions.

Many aspects of the encoding make sense to me, but many others don't!

We have a bitwise operation that (in non-obvious ways) can give us all the classics: A or B, not A, etc.

We have a bitreverse operation along with popcorn and count leading zeros.

These look like eg

bitrev (Reverse Bits)

47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0		
?	Dx	Ax	?	A	0	D	Dt

See how we bits 39..28 are all zero. Once again I *think* that's basically a "don't prefetch" setting, and we can set a different register specifier in there to prefetch a single register for a later operation.

What about floating point?

fmadd (Floating-Point Fused Multiply-Add)

39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0			
Bm	Bt	B	Am	At	A	L	D
63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	Dt S	1 1 1 0 1 0			

?	Dx	Ax	Bx	Cx	?	Cm	Ct	C
---	----	----	----	----	---	----	----	---

The main new feature is that each operand is modified by two bits (Am, Bm, Cm). One of these bits takes the absolute value of the input, one negates the input, so we can have any of A, -A, |A|, and -|A| as an input.

Also, I believe that because of how registers are specified, we can

- take in a mix of 16 and 32 bit registers via the register specifiers
- perform the calculation in 32 bits (that's the meaning of this particular operand, that it uses the FP32 pipeline)
- store the result in 16 bits

In other words you can pretty much mix and match these as you wish. Many mix and matches are

stupid, but some may make sense, like taking FP16 multiplier inputs, but maintaining a running total in an FP32 accumulator (meaning FP32 C and D).

There's a companion

fmadd16 (Half Precision Floating-Point Fused Multiply-Add)

39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
?	Bm Bt	B Am At	A L	D Dt S	1 1 0 1 1 0
63 62 61 60 59 58 57 56	55 54 53 52 51 50 49 48	47 46 45 44 43 42 41 40	?	Dx Ax Bx Cx Cm Ct C	?

which is much the same expect note that At, Bt, and Ct are now three, rather than four, bits long.

I think the missing bit is the length/immediate control bit, meaning that

(a) if you want to use the FP16 pipeline, the input values flowing in *must* be FP16 values.

(b) you cannot set an immediate as an input specifier

So essentially the highest At bit distinguishes between lane and uniform registers, and for lane registers we have two cache control bits, while for uniform registers we have one bit additional register specifier bit and one bit that may (or may not, unclear!) hint that the uniform value should be Operand Cached.

Then we have a bunch of simple single operand instructions like

floor

47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
?	Dx Ax	0 0 0 0 0 0 0 0	Am At	A L	D Dt S	0 0 1 0 1 0

and

trunc

47 46 45 44 43 42 41 40	39 38 37 36 35 34 33 32	31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0	
?	Dx Ax	0 0 0 0 0 0 0 1	0 0 0 0 0 0 1 Am	At A 1	D Dt S	0 0 1 0 1 0

At first sight there's nothing surprising here, but the encoding does seem kinda dumb. We have a primary opcode, then what looks like a secondary opcode of 4 bits in the range 31..28, then more secondary opcode in the bits 32 and 33; and perhaps also use of the L bit as a secondary opcode. But this primary opcode seems to ultimately control 13 instructions (along with above some things like reciprocal, transcendentals, and two graphics specific dfdx and dfdy instructions). That means you could code them all using bits 31..28, and use the L bit as a length bit, avoiding the use of the instruction extension two bytes whenever Ax and Dx are zero (ie we're not using very many registers)!

Once again I don't fully understand the decisions made here.

Missing is divide, which you can hopefully fake (well enough for GPU purposes) via reciprocal followed by multiply.

Then we have some flow control instructions: call, return, branch, trap, halt; and some somewhat related predicate manipulation instructions.

Then we get onto instructions that looks monstrous, but now that we understand how the system works, are not that bad. For example

icmpsel (Integer Compare and Select)

39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
?	?		Bt		B		?	?		At			A		L		D		Dt		0	0	1	0	0	1	0	0	1	0	0	1	0	0	1	0			
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40
?	?	Dx	Ax	Bx	Xx	Yx	?	?	?	cc	Yt		Y	?	?	?	Xt		X																				

So basically compare A to B and based on the comparison, set D to either X or Y.

What does “compare A to B” mean? That’s defined by the three bits cc which determine various types of comparisons: less than, equal, possibly with signed extension vs unsigned if we are comparing a 16b value to a 32b value.

We have something similar for floating point, now with the ccc bits not establishing signed vs unsigned extension, but establishing how to handle or ignore NaNs in the FP comparisons.

Then shuffles and reductions. Interesting instructions, but nothing new in their encoding.

load/store instructions

Now look at this instruction:
`wait`

`wait_for_loads()`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
?	?	?	?	?	?	?	?	i	0	0	1	1	0	0	0

Dougall doesn’t tell us anything about this. But one can imagine possibilities.

I earlier talked about the issue of instruction to instruction dependencies. Within a clause, dependencies are detected at Decode time, and are handled by ensuring that a successor instruction launches at least n cycles after some predecessor. But what about dependencies across clauses? I suggested this was handled at a very coarse granularity, something like clause $N + 1$ is required to wait until clause N completes before it begins. The only case in which this is really interesting is the case of dependencies on loads.

It certainly looks like the above wait instruction is how this idea is implemented. Maybe something like, at the end of a load clause this is the last instruction. It does a preliminary check to see if all the loads have completed and then completes. Otherwise if there are loads pending to L2 or further away, it sends a message to the general thread controller to swap this warp out from the active list, and swap in another warp.

Other things we learn by looking at the memory instructions are that there are separate load instructions to load from tiles, from device (ie “main”) memory, from the stack, and from Scratchpad memory. One interesting implication of this is that, unlike nVidia, you cannot just store a raw pointer in some sort of complex data structure (hashes, trees, whatever), then extract it and dereference it. You have to know what the pointer points to and that isn’t obviously defined in the pointer. (Maybe it is in the top few bits, like nVidia does it, but you’re certainly not supposed to look around at these top bits to decide which load instruction to call! Perhaps we will see this evolve if Apple tries to move GPUs to more

flexible GPGPU use cases, as seems to be their goal.)

Device memory loads load into one to four successive registers. So we need to start with a destination register specifier.

device_load

The destination is register R. It is specified somewhat differently from register D because, as we now know, loads execute in a separate, non-Datapath clause, and don't have a connection to the Operand Cache (and in fact write to the Register File via a different path involving a separate Write Queue, etc, as we have discussed). So you will see that the R modifier bit Rt is a single bit, which I assume is a length specifier (32 vs 16 bit destination registers) with no cache hint bit.

Next we need an address register to read from, and that's specified by the A register, split (who knows we have a different encoding) into the 4 Ah and 4 Al bitfields. The At modifier bit cannot modify length (address must be 64b) and cannot affect caching, so what does it do?

It defines whether we get the address from either a uniform register or a per-land register. Either way, we need to use an even-aligned pair of 32b registers, and so one of the 8 address bits is redundant (there are 64 “64b” per-land registers and 128 “64b” uniform registers). Maybe Dougall is misinterpreting how bit 16, the lowest bit of Al, is used?

To this base register (either uniform or per lane) we can add an index. This is the offset register, encoded again split into Oh and Ol bits. More specifically, start with modifier bit Ot.

If this is 1, then the input specifier is an immediate, encoded as 8 high bits in Ox, and 8 lo bits in Oh:Ol.
If O_t is 0, then the offset is a 32b register. Based on O_u it's either a uniform register or a per-lane register.

So now we know that we are loading from A+O into R. How much data are we loading?

That's indicated by mask, which loads a value for each bit set. So if all bits are set, then four values are loaded. If we have a mask like 1011 the we will load essentially address[0], address[1], and address[3] into R0, R1, and R2; so we can skip over elements, which may be convenient if we are loading every second element of an array, or if we are loading something like just the RGB values from an RGBA array.

Somewhere in this we also need to specify how the data is to be interpreted and unpacked (remember that load can do all manner of fancy transformations of various different RGBA formats and orderings). This is presumably indicated by the remaining bits, but we don't yet know? (Maybe the Asahi graphics team know?)

The fact that the load/store instructions have such different encoding makes one wonder if decoding is in fact a distributed process.

We saw that the very first patents for clauses and L0 caches had instructions decoded between L1 and L0; and that patents slightly later changed this to decoding between L2 and L1 (which of course has the downside that the L1 is effectively smaller, but the upside that decoding once done can persist a lot longer than a decoding stored in a clause cache.

Perhaps the optimal end point is a first round of decoding from L2 to L1 that establishes things like the length and class of each instruction, and then a second round of decoding from L1 to L0 that is optimized (and uses an optimized bit format) for the use of each particular execution unit. So the Datapath decoder has machinery for worrying about cache hints and extra prefetch registers, while the Load/Store decoders can avoid that machinery while, for some reason, seeing advantage in a different way to encode the address and offset registers?

There are similarly mysterious variants of this for storing to device memory, and reading or writing from other locations. The most interesting of these is

uniform_store

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
L	b	s	Rx	0	0	0	0	Oh	?	?	1	1	1	unk	Ot	OI	0	0	0	0	R	0	F	1	0	0	0	1	0	1	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	

This writes a single value (16b or 32b) to register R (8 bit specification, with Rt indicating 16b or 32b destination).

The source is register O. Once again modifier Ot indicates whether we write an immediate value (Ox:0-h:Ol) or ...

Or what?

Maybe the other possibility is a uniform, so this give a way to move one uniform to another. But that does not seem very useful, at least not until we have a full suite of uniform modifier operations. Right now we have no uniform operations, just storage, so what's the point in storing the same value in two places?

It might be convenient to read a value out of a lane and store it as a uniform, but if we do that, which lane would we choose? Perhaps some combination of the various unknown bits (eg maybe the five bits 55..51) define from which lane to read the value?

Of course there are many more instructions, eg various graphics specific instructions, or atomics and barriers. But this gives us a flavor of how the ISA encoding works.

Permute Instructions

I've left permute instructions till the end because it's only after seeing and thinking about other instructions that you appreciate the complexities these imply.

(2013) simple quad cross-lane instructions (dfdx , dfdy)

Let's start with (2013) <https://patents.google.com/patent/US9183611B2> *Apparatus implementing instructions that impose pipeline interdependencies*

This included a figure we have already seen,

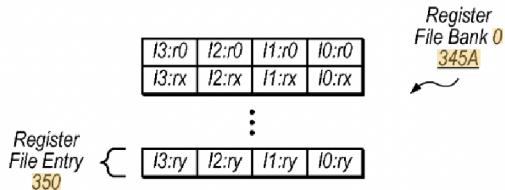


FIG. 3A

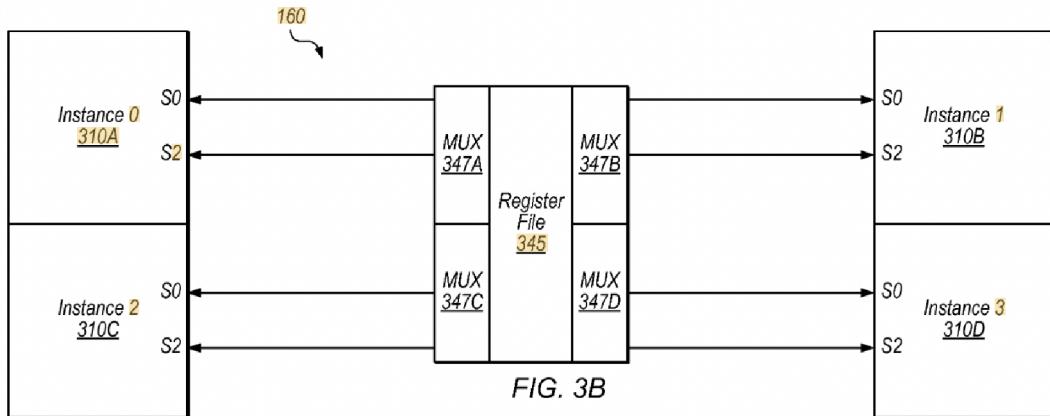
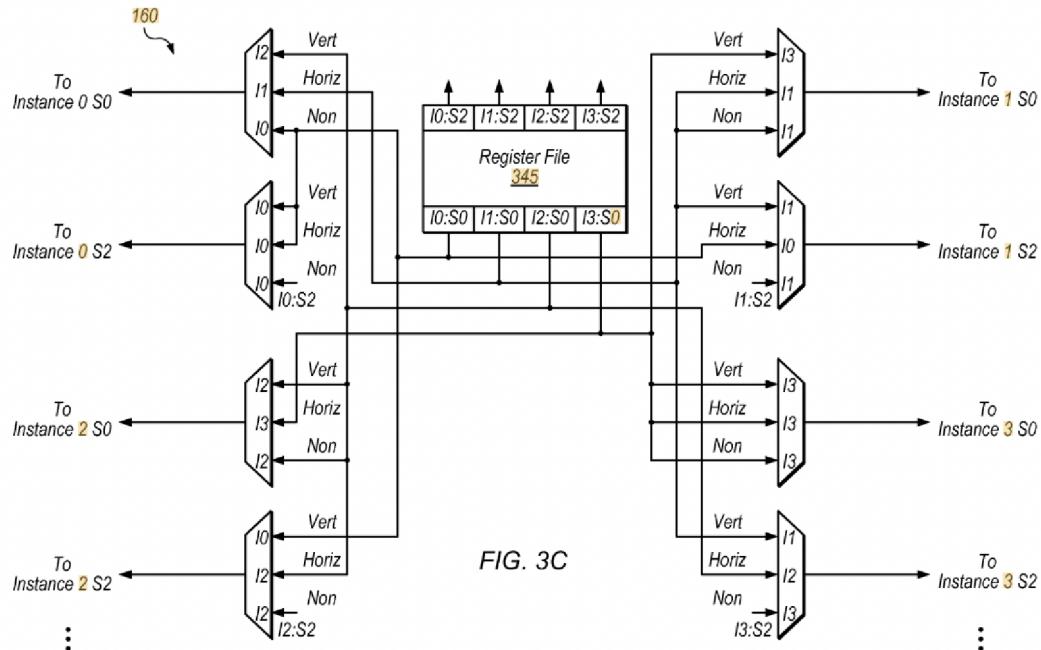


FIG. 3B

Physically each set of four execution lanes was (at this time) wired to a central register file holding four lane's worth of registers.

GPUs need to provide instructions that act on 2×2 quads of pixels, eg to give a partial derivative approximation for the quad (ie $\partial f / \partial x$ or $\partial f / \partial y$). These instructions, generally called something like `dfdx` and `dfdy`, require inter-lane operations.

The patent says that, based on the way the lane registers are stored across register storage (essentially as 4 contiguous registers for four lanes, all read at once from the bank), a routing mux between the register storage and each lane can handle the job (usually sending same lane to same lane, but with some degree of lane permuting possible).



In other words, to handle the specific needs of `dfdx` and `dfdy`, place some routing muxes between the register file and the execution units.

The patent specifically points out the obvious fact that this won't work the an operand cache, and says that for the gradient instructions, the operand cache is bypassed.

Apple provided quad-scoped permute instructions with the A11 (shipped 2017). There doesn't appear to be a patent for this, and the obvious assumption is that these permutes were implemented in this same way, via routing on exit from the 4-wide register file, and bypassing the operand cache.

(2019) A13 one-operand permutes

With the A13 (shipped 2019) we get full 32-wide permutes, so how are those implemented?

(2019) <https://patents.google.com/patent/US11294672B2> *Routing circuitry for permutation of single-instruction multiple-data operand* tells us this is now done by routing an operand through a two-level crossbar. The first level is a 4 to 4 crossbar, which by itself can handle all the quad permutations; the second level then handles permutes between quads. To simplify things, the second level permute network can only handle some simple (but common) permutes; more complex permutes require multiple passes through the second level network.

We are not told anything more, but obvious assumptions would be that

- the register file and operand cache are now as wide as is most convenient, so probably 32-wide?
- this permutation network sits at the exit point of the operand cache, so one exiting operand can be routed through it
- quad permutes save power by not activating the second level network.

(2019, 2020) A14/M1 matrix multiplication

The A14/M1 gain matrix multiplication.

A consequence of having this general permute is that, for better or worse, Apple can implement matrix multiplication within the general FP execution unit rather than adding a tensor unit. Metal hides this behind a single instruction, but (2019) <https://patents.google.com/patent/US11126439B2> *Datapath circuitry for math operations using SIMD pipelines* gives the full horror of how the data are routed to perform the successive steps of the matrix multiplication.

The main new thing in this patent is that it suggests that matrix multiplication requires *two* shuffle networks, to rotate two operands, ie both input matrices. Fortunately this second shuffle network used (so far anyway...) only for matrix multiply can be a lot simpler.

The matrix multiply instruction (in one particular case, details will differ depending on FP32 vs FP16, how many FP32 units are available [A14 vs M1], etc) expands into a sequence of microcode that looks like

- o f16fma tmp0.16, R12L, R13L, R8L.x
- o f16fma tmp1.16, R12L, R13H, R8H.x
- o f16fma tmp0.16, R12H, R13L, tmp0.16
- o f16fma tmp1.16, R12H, R13H, tmp1.16
- o f16fma tmp0.16, R12L, R13L, tmp0.16
- o f16fma tmp1.16, R12L, R13H, tmp1.16
- o f16fma tmp0.16, R12H, R13L, tmp0.16
- o f16fma tmp1.16, R12H, R13H, tmp1.16
- o f16fma tmp0.16, R12L, R13L, tmp0.16
- o f16fma tmp1.16, R12L, R13H, tmp1.16
- o f16fma tmp0.16, R12H, R13L, tmp0.16
- o f16fma tmp1.16, R12H, R13H, tmp1.16
- o f16fma tmp0.16, R12L, R13H, tmp0.16
- o f16fma tmp1.16, R12L.x, R13H, tmp1.16
- o f16fma R15L.x, R12H, R13L.x, tmp0.16
- o f16fma R15H.x, R12H.x, R13Hx, tmp1.16

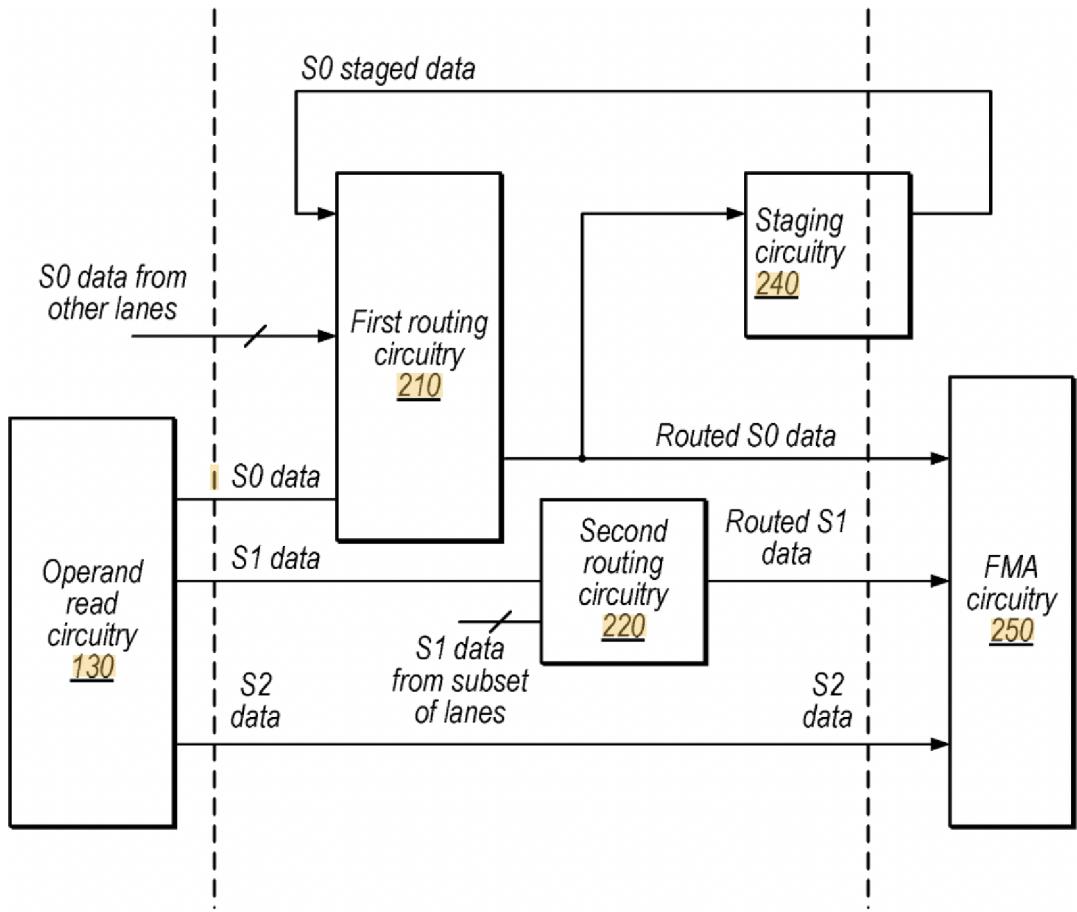
Omitted from this is shuffles that are applied from one cycle to the next to the inputs.

Some things that are clear are

- the ability to expand single instructions to microcode is actually of some value!
- we need two temporary registers,
- we're repeatedly using the same inputs but shuffled each time.
- performing data rearrangement inline with the FMA's means we can just run an uninterrupted stream

of computations without losing any cycles to data movement

This leads to a design that looks like



The part we had before, for permutes is S0 (source 0) data flowing into First Routing Circuitry.

New is

- Staging Circuitry. This holds one of the operands (actually R13 in the above) so that it can be repeatedly permuted to keep generating new operands for the FMA circuitry. I assume providing this temporary storage saves energy by giving an optimized re-read path (and the pattern of shuffles, if applied to already shuffled lanes, may be lower energy?)

The other operand is subject to simpler shuffles. The diagram does not show temporary storage but I

suspect that may be for simplicity; the same reasons that justify Staging Circuitry 240 would seem to justify an S1 Staging Circuitry.

- presumably there is also some storage for the temporaries.
- this same patent at the end, also discusses reductions. These likewise consist of microcode that repeatedly shuffles data and performs an operation. The temporaries that are used for matrix multiply could be, and probably are, also be used for reductions.

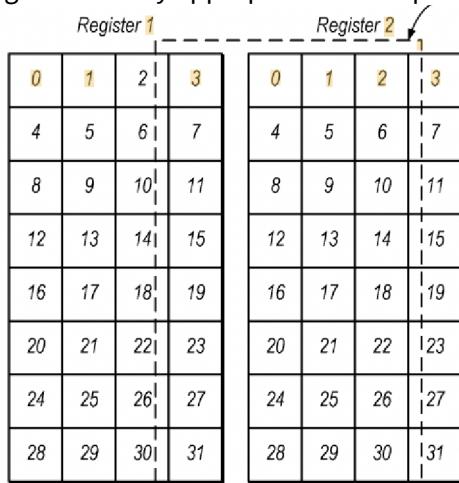
(2019, 2021) A15/M2 shuffle-and-fill permutes (two operand)

The A15/M2 gains the so-called Shuffle-and-Fill instructions.

This is described in (2019) <https://patents.google.com/patent/US11126439B2> *SIMD operand permutation with selection from among multiple registers*. The obvious point of these instructions is to be able to perform more work “in-SIMD” without having to resort to additional loads. Consider for example some sort of convolution/filtering. This will involve moving a mask over pixels. If you have loaded one group of pixels into a first register rA, and the adjacent group into rB, then it would be convenient to somehow be able to slide the filter window to straddle both rA and rB. Another way to think of this is it’s a SIMD-level version of the CPU bit operation often called funnel shift.

There are fancier options to this instruction which allow it to also manipulate 2D data. We won’t explain these, but a diagram shows the idea:

Here R1 and R2 are holding data to be interpreted as 4x8 image blocks, and we wish to extract the “geometrically appropriate” overlap for eg a filter operation.



Obviously this can’t be achieved purely with a permute operation, and the patent doesn’t tell us how it is achieved, just how the instruction is useful (and can easily be implemented in the 2D version, at least if your image blocks are power of 2 wide).

Doing a full permute that can take in 64 (rather than 32) values seems unlikely, too much extra hard-

ware!

Maybe we can do it by adding a very simple select mux before the two level permute? What we really need is simply the value in each lane that will appear in the final output, regardless of their order. For each of lanes 0..31, a mux chooses the value from either operand A or operand B, and then the resultant mess goes to the permute network as a single 32-element operand which permutes it into the correct order? So, looking at the above diagram, this initial layer will route lanes 3, 7, 11, 15, 19, 23, 27, 31 from register 1, and all the others from register 2, then the permute will rearrange them as required. Another way to do it would be if we can override the predication network. Run the permute machinery once on register 1 and write the result to the operand cache, then run it again for operand 2 and write it (with a “fake predicate mask”) to the operand cache so that only some value get overwritten. This second option is, as far as I can tell from the patents, not what is done; it might be a little slower, but also might be valuable for future other purposes?

As far as I can tell, this aggressive flexibility in cross-lane permutations is something unique to Apple GPUs. This means it's the sort of thing that will require GPGPU enthusiasts to think about how best to use Apple's specific HW, rather than simply copying over a pre-existing CUDA-optimized algorithm. It could, for example, substantially accelerate small matrix transpose.

nVidia, for example, has shuffle operations that can handle a few common cases (eg funnel shift, needed for filtering; or butterfly, needed for reductions) but not general permute.

Looking at this, you can also start to think of fusion type operations... For example you could imagine fusing a permute with the next instruction that uses the result, so that the two operands flow out of Operand Cache, one gets permuted, the add or whatever happens, and we write back the result. This would, for example, allow third parties to write specialized matrix code (at the very least complex matrix multiply) matching essentially Apple's speed (code that looks like a sequence of permutes and FMACs, where the permutes are all fused with the succeeding FMACs).

Similarly the first selection stage I'm suggesting is used for the Shuffle-and-Fill operation could very easily be turned into a condition-based Select, that could then be fused with the next instruction. GPUs are maybe not yet close enough to their limits that aggressive fusion makes sense. But one day?

QoS

As always, along with simply scheduling and execution, we want to impose some sort of QoS mechanism so that, when resources run low, appropriate tradeoffs are made between different clients. Things start at the OS/driver level, by submitting commands to the driver.

Even in the very first patents we see the presence of multiple command queues (one per application?) We've already seen how important multiple command queues was for nVidia, in that before multiple command queues were present, the theoretical ability to schedule multiple kernels in parallel did not mean much without the ability, to examine the heads of multiple queue to find appropriate simultaneous loads.

(2015) Basic framework for priorities and enforced sharing

We start with (2015) <https://patents.google.com/patent/US20160358305A1> *Starvation free scheduling of prioritized workloads on the gpu*. This lays out the basic problem: At times of GPU stress, we may have multiple applications, each with multiple command queues, submitting commands to the GPU (through the OS/GPU driver). When things get tight, which commands will we drop?

At this stage (pre 2015) we have limited special hardware to help us out, so what can we do at the driver level? We use ideas from operating systems.

Each application has a *static* priority, and we walk the list of clients in priority order, estimate the resources required by the client, and build up a list of potential work until we believe the resource budget for this frame (essentially the amount of time available) has been exceeded.

Of course if this is all we did, some clients might be frozen out forever, so also have a mechanism to slowly raise the *dynamic* priority of unlucky clients.

The final piece required is rough feedback from the GPU when a kernel takes longer than its predicted budget, so that the budget predictions stay at least approximately accurate.

The static priorities can be set in various ways. At this early stage they were more or less hardwired, so that GUI was highest level, camera came next, with “normal” applications (browser, games, etc) next, and as usual background utility apps come last.

Just as with operating systems, there are a number of precise details you can imagine for tweaking this scheme, but the specific point they care about is that clients may shift relative priority within their band, but the highest level band is always more important than the next level, and so on down. In other words Safari and a game can fight out who gets what’s available of the GPU, but the GUI will always get as much as it needs, before Safari and the game get anything.

There are other particular constraints, for example if kernel take too long (in absolute terms, or relative to the budget they were allocated) they will be killed, to try again next frame, hopefully in a more sharing mood!

These constraints matter because, remember the point we keep making, that context switching on GPUs, until recently, was not to be taken for granted; you couldn’t just interrupt a long-running kernel to do some light-weight GUI work.

There are specific details that are of interest; for example if one kernel goes over budget, every kernel associated with the app will be demoted. If you think about it, what these precise details mean is that if an app does a bad job of throttling itself relative to other apps it is that app that will stutter and look bad every frame, but the app will have limited ability to drag down other, better behaved, apps.

(2018) split large compute tasks into smaller units

We’ve already seen the 2017 patent to try to handle priority inversion by boosting GPU frequency when a high priority task is stuck behind a low priority task. But that idea clearly can only slightly improve the

situation.

By 2018 GPU compute (in particular things like CNN's utilized in image recognition AI) was becoming important, with the potential to block the GPU for substantially longer than a frame. (Think of the GPU running image recognition on newly imported photos.)

So we get (2018) <https://patents.google.com/patent/US10908962B1> *System and method to share GPU resources.*

The important idea here is to ensure that at least the elements of this code controlled by Apple (think, for example, of Metal Performance Shaders) are written to limit the length of time spent in a kernel between "yield points" so that an MPS kernel can be gracefully swapped out at a yield point, should that be required. (I think, at least in this first implementation, this was done by breaking up a full MPS kernel's grid into multiple, separately submitted, sub-grids, of limited size.)

An additional concept mentioned in this patent is the use of "blacklisting". The idea is that if GPU compute has to be interrupted by "real" graphics, the GPU compute kernels are frozen in the firmware submission queue for a particular period of time (the patent suggests 4ms) before being allowed to advance to the GPU, so that during this period "real" graphics can proceed without resource competition.

This raises an interesting point. Although many things have changed since this 2018 patent, it remains the case that long-running compute work may frequently be interrupted by GUI processing, which seems sub-optimal. An obvious solution presents itself, which you can claim as being inspired either by E- vs P-cores, or by iGPU vs dGPU, namely to provide, alongside the "serious" GPU (call it the dGPU) a lightweight GPU (call it the iGPU) to handle lightweight GUI work. This would ensure that at least in the common case where one program is making serious compute use of the dGPU (for physics, simulation, AI training, ray tracing, whatever) that dGPU will not lose performance (both from context switching and constant cache overwrites) to a steady stream of minor chatter from the GUI as it needs to update the screen 60 or 120 times a second. Obviously Apple have had OS code for some time to make the decision as to whether to use the dGPU vs iGPU for any particular task, so a solution like this does not have to start from scratch; much of the work can build on the experience of Intel Mac graphics.

Alternatively, perhaps you could achieve much of this by simply locking some number of GPU cores (perhaps 1 core on A# or M# devices, 2 cores on a Pro or Max device), along with some fraction of the L2 cache space, to dedicated GUI, nothing else, so that there are limits to how much GUI work can perturb other GPU work?

(late 2018) hardware counters and two-tier submission

With a later 2018 patent, <https://patents.google.com/patent/US10795730B2> *Graphics hardware driven pause for quality of service adjustment* we start to see a more sophisticated design. The two important elements that have been added are

- We now have many dedicated counters in the hardware, counting many different things, so we can make more informed decisions
- Because of this better data, we have insight into which particular parts of the GPU at any given time are being overwhelmed

The specific way this plays out, as described in the patent, is that in the past we were essentially giving kernels a certain number of milliseconds on the GPU, and punishing them if they exceeded that. But we can now see whether the part of the GPU that is being stressed is vertex processing, fragment process-

ing, or compute processing. Based on these, we do things like throttle back the rate at which vertex kernels are being submitted to the GPU, while continuing to send compute and fragment kernels as fast as possible.

The previous priority system seems to be somewhat retrofitted onto these new capabilities. The idea seems to be that, at any given time, one of these three services (Apple calls them *data masters*) vertex, fragment, or compute, is designated the highest priority (based on the previous priority scheme). That “primary” data master is allowed to flow at full rate, while the other two data masters can use whatever GPU bandwidth is left over.

The GPU tracks if the primary data master is forced to stall (ie there are periods during which it wants to submit a command to the GPU but cannot) and if that happens too often, the non-primary data masters will be held back a little so that they stop crowding out the primary data master.

(2020) multiple per-app command queues (rate normalization)

The initial Metal design essentially made a
and a process synonymous – one queue per process.

However this changed over time. The timeline seems to have been that there was a desire for indirect command buffers (ie the ability for code executing on the GPU to submit further code without requiring the CPU). At this point you then have at least two command buffers of a sort, one from the CPU, one from the GPU. At which point it makes sense to also allow for multiple command buffers on the CPU, perhaps to allow different parts of the code independently to submit GPU tasks.

This is all fine, but it means that the precise details of all the previous QoS code have to be modified. Otherwise an app could grab most of the GPU time for itself simply by using a large number of command queues, each of which gets an equal share of GPU time!

So, most importantly, now instead of allocating resources at the level of the command queue
- resources (and stalls) are *tracked* at the level of the command queue but
- resources are *allocated* at the level of the process.

The per-process resource allocation is then split into per-queue allocations within that process, after which everything happens as before, including tracking that each command queue is staying within its allocation, and occasionally pausing low priority data masters if higher priority data masters are being stalled too often.

This small tweak to the system is described in (2020) <https://patents.google.com/patent/US11321134B2> *Normalizing target utilization rates of a cross-application table of concurrently executing applications to schedule work on a command queue of a graphics processor*. (The title is a mouthful, but more or less explains the point of the patent.)

(2021) <https://patents.google.com/patent/US20230075531A1> *Quality of Service Techniques in Distributed Graphics Processor*

As we have seen repeatedly with Apple, after evolving a system for some time, the best elements of the

system are retained, but placed in what looks like a completely different system. We see this in a collection of related patents all filed in 2021, of which the above is a representative example. These patents describe a completely new, QoS-based, scheme for handling kicks, from submission by firmware through breaking grids up into batches, through tracking degree of completion. The full details of his new system for dispatching graphics remain unclear, but some elements are visible. The new scheme is described in terms of an M1 Ultra style design but

- it's likely that the basic design is used even for non-Ultra hardware
- it's unclear if this design is actually present on the M1 Ultra, as opposed to being a design goal for newer hardware (like perhaps an M3 Ultra)

To start with, Command Queues with Processes are now described as having two QoS properties

- a Priority (which essentially describes ordering, ie which Kernels get executed first) and
 - a Weight (which essentially describes what fraction of the GPU's "width" a kernel should consume).
- Priority probably cannot be controlled directly by the app, but Weight can, so that an app can describe that it wants to allocate say 70% of the GPU (ie 70% of the GPU resources the app is allocated) to Queue A, and 30% to Queue B.

A second change is that, just as we saw a slow shift from things like power and DVFS being controlled by the OS to being controlled by dedicated hardware, so it seems like what was being done by GPU firmware is now slowly being moved into dedicated hardware, along with the presence of many more counters. (To some extent these go together, since once too many counters are involved, it's inefficient to have generic hardware running firmware trying to process them all.)

So we begin with the sort of design that we discussed earlier in the *Affinity-based Graphics Scheduling* patent: processing streams of commands (separated by pre-and post-processing work, and tracking dependencies between kernels). The result of this work is a stream of kicks (essentially kernels) which are split into work batches to be sent to "the GPU".

The innovation at this stage, already discussed, is that each work batch can be annotated so as to be scheduled to a particular hardware. The obvious way to do this is something like saying either "execute this on GPU core 3" or "execute this on GPU chiplet 0", but in fact the scheme is more like "execute this on the same core [or chiplet] that executed a previous batch labelled N" which is slightly more flexible since the first level of scheduling only has to care about how wide a kernel should execute (core-wide?, chiplet-wide? entire GPU-wide?) along with sequencing, while leaving precise placement on each core to the second level of scheduling (which will do so based on load-balancing). This scheme is described as the first level scheduler placing the work batches in *virtual slots*, which are materialized by the second level scheduler to *distributed hardware slots*, generally called *dslots*, executing on GPU cores generally called *mGPUs*.

A dslot essentially refers to the exclusive occupation of a GPU core (and most importantly of its Local Address Space) for some period of time. Because the Local Address Space, and the variant ways it can be used, for example as a Tile of Image Storage, multiple kernels within the same kick that all have some sort of agreement as to how they are supposed to share this Local Address Space and Tile can

work together and be scheduled together in the same slot, but a random kernel from a different kick can not.

The patents are very vague on details, but the idea seems to be that a slot is conceptually a timeslot within the barrel-processor/time-sharing model of a GPU, but with the constraints I have mentioned that different slots executing more or less at the same time on the same core have to be part of the same kick so that they understand that they are sharing the Local Address Space/Tile.

If we go outside the world of graphics to the world of compute, you could imagine eg two different kernels, each of which only use a portion of Local Address Space, could maybe be allocated together, with some lightweight virtualization of the address of each line of Local Address Space, but this does not appear to be part of the design plan.

mGPUs are gathered together into *Groups* which can essentially be considered chiplets, large aggregations of GPU performance that can share data, but not as easily as handling data within the single chiplet/Group.

The kicks flow through the sort of processing we have seen before:

- first each kernel has its dependencies tracked, along with when the kernel completes. Once all dependencies are resolved, priority and age are used to determine the sequencing of the kernels in a queue of upcoming work
- next the kernels are split into batches of workgroups
- which are then dispatched (according to the allocation masks) to physical hardware

The primary conceptual change is the conversion of what were physical queues of kernels and batches into virtual queues of batches which are only materialized at the last possible stage, allowing for better load balancing. The virtual queue scheduling is used to ensure ordering and priority; the physical dGPU scheduling is used to optimize for affinity (try to schedule work where earlier related work was done), after which we try for load balancing.

There are hints that (as I suggested earlier) some dGPUs may be permanently reserved for lightweight but high priority GUI work. Details are, of course, unclear, including whether this is now policy for all GPUs, or specifically a feature of Ultra-sized GPUs (ie those with a large number of cores/dGPUs).

Within this framework, I'd be the first to admit that there remain many parts that are unclear. For example the exact way successive kernels working on the same tile are linked together is never specified; it's never even suggested that this must be a concern. Similarly, I've mentioned how there appears to be no way to share Local Address Space, resulting in constraints as to when kernels can be scheduled together. This seems suboptimal and something to be addressed. A constant problem I have with the GPU is that, unlike the CPU, I don't have a good feeling for how ambitious the team are, how aggressive they are in seeing sub-optimality and working to remove it. An example of this sort of inefficiency is seen in the "slot retention" procedure. This is a scheme whereby, once a dGPU has completed its slot of work, the dGPU is not immediately available for reuse; rather the dGPU may be placed in a "retained" state while SW or FW query performance registers and suchlike. It's only after this is complete that the dGPU slot is released.

While understandable, this seems like the sort of thing that would have been squelched out of the CPU years ago; it seems like a bad idea to slow the reuse of an entire GPU core just for the sake of a few basic registers! Rather do something like duplicate the registers, then at reuse time switch to the second set and move the first set out to some appropriate location; details can vary but the basic point is that large expensive hardware should be working every single cycle, not being blocked for multiple cycles by silly minor issues. Perhaps the assumption is that this is a rare case, only relevant when profiling, and so not worth optimizing for?

As mentioned earlier, within this new design the splitting of grids into batches is now done with an eye

to affinity. The idea seems to be that a grid is first split into large portions by L2 affinity (so two portions on an Ultra, one portion on a smaller chip) then each of these L2 portions is split into L1 batches (presumably using the same heuristics as we've already seen, so that batches try to overlap maximally along x, y, and z axes to maximize edge overlap reuse). Practically this doesn't seem to change much for designs smaller than an Ultra, but perhaps in future we will see something like a smaller L2 associated with say groups of 6 or 8 GPU cores, and these L2's communicating via a GPU L3? In which case there would be value to such L2 affinity scheduling even on non-Ultra designs. The one part that does seem to be notably different from the previous design, even on smaller GPUs, is an implementation of work-stealing, ie the ability to steer workgroups from their "designated" mGPU (based on cache affinity) to a different GPU if that GPU finishes earlier and is now idle. But, as with much of this patent, the details are unclear as to just how ambitious and powerful this work-stealing is.

We have already seen a few places where QoS and weights can be implemented in the scheme, eg by where full kernels are placed in the "pending" execution queue, and how many, and how often, batches are dispatched from a particular queue to some or all of the available GPU cores. This QoS scheme is now unified at all levels and extends down to at least the decision as to what workgroup next to move from inactive to active within a core. The QoS may even extend down to which of the currently active cores to schedule next; the patent is unclear on this point, but perhaps that's too fine-grained (in terms of transistor overhead) to be worth doing?

Two different types of "billing" are tracked, for the purposes of QoS (and perhaps at some point for other purposes), namely "activity" billing (how many cycles a kernel executed) and "opportunity" billing (how many cycles a kernel was active in a GPU core, but didn't execute for whatever reason, eg was waiting on a register read); the difference between these is an indication of inefficiency. Right now the primary use for counting stalls seems to be that if a high priority kernel is stalled too much by a co-executing lower priority kernel, there's an assumption of interference between the two (eg both are constantly reading registers that are not in the operand cache), and the lower priority kernel is removed from co-execution, or at least throttled back, reducing GPU occupancy. At some point, though apparently not yet, a more granular set of such performance data might be usable to increase efficiency; for example a kernel that's frequently waiting on the texture unit might be co-scheduled with a kernel that's rarely uses the texture unit. Right now at attempt at this is done, but only at the highest level of vertex vs compute vs fragment. Better performance data would also allow for a better choice of co-executing non-interfering low priority work with high priority work, rather than the current rather blunt scheme of just throttling all low priority work if the higher priority work stalls too much.

So far we have seen "instantaneous" machinery (eg feedback about stalls) that allows for trying to hit a certain level of QoS in a *reactive* way (ie after things have happened), but even better would be to load balance in a proactive way, which is done by maintaining statistics across prior batches, and using those statistics to make future decisions, and this is also implemented to some extent.

The final part of all this QoS machinery is one more attempt to deal with priority inversion where, either I don't get it or it's very obvious; but the idea seems to be that

- the distributed system can indicate when a high priority kernel is queued behind a low priority kernel
- an attempt can be made to move the high priority kernel and, if that's not satisfactory

- a context switch can be forced.

Presumably these basic ideas are, to some extent informed and modulated by collected statistics (for example if we believe the earlier kernel will complete soon, then we might as well at least wait for that completion before forcing a context switch) but this is just one more unclear detail in a substantially opaque patent.

As a final addendum to this discussion of Ultra-style designs, we have (2021) <https://patents.google.com/patent/US20220237028A1> *Shared Control Bus for Graphics Processors*, which is another detailed NoC design, this time for the *GPU control* NoC between multiple Ultra-style chips. There is a separate NoC (presumably an extension of the third/fourth generation NoC we have discussed elsewhere) for communicating data between chips.

At the highest level, this control NoC is responsible for communicating the various elements we have already discussed – workgroup batches from a global distribution center to various distributed execution units (ie the various chiplets of an Ultra design) along with feedback in the reverse direction as to progress and completion. My guess is that, like the above Ultra design, this is not actually implemented on the M1 but is, rather, a more sophisticated design for the M3 and later Ultra's.

In this NoC we see all the usual Apple concerns and solutions: QoS and priority, credits for flow control, use of an NoC (ie a network, with packets and routing) rather than a simpler bus-style solution, and the use of a solution that's specific to the task (ie with its own particular rules for memory ordering and everything else, rather than a one-size-fits-all solution general purpose NoC). This particular NoC cares about packetization in a way I have not seen for the other NoCs, I think because the control data to be transferred (details of each workgroup batch including register setup) can be very variable in length, unlike the other situations (like snooping, or memory requests) on the primary per-chip NoC.

This particular NoC is linear (so either a chain or a ring) which means, of course, that it can scale, fairly easily from a two-element Ultra to a four- or even eight-element design without too much difficulty. More precisely you can view the overall scheme as two-level network, one level communicating chiplet to chiplet, the second level communicating core to core within the chiplet. Right now both of these appear to be chains; presumably at some point both levels will be upgraded to rings.

Another interesting scalability point is that whereas the current M1 Ultra design (and even some of the earlier patents which seem to be post-M1) are based on a single global distribution center, this patent suggests the presence of multiple such global distribution centers (and of course each chiplet of an Ultra will have such hardware on it). The idea seems to be that an individual chiplet can be controlled only by a single global distribution center (which may control multiple chiplets), but we could have say one such global distribution center controlling three chiplets, and a second global distribution center controlling a fourth chiplet. It's unclear quite how this might be used, but one can imagine a few situations, for example either at the hypervisor level, or devoting one chiplet to UI and mundane graphics, while the other chiplets handle a large AI computation. This scheme would work better if a chiplet could be broken into finer granularity, and that may be coming – the patents are always somewhat ambiguous as to whether the “smallest” unit of relevant control is an entire chiplet or something smaller like, say, 10 GPU cores.

There's something of an update to this patent in (2022) <https://patents.google.com/patent/US20230419585A1> *Tiled Processor Communication Fabric* which gives more details of the fabric. Don't

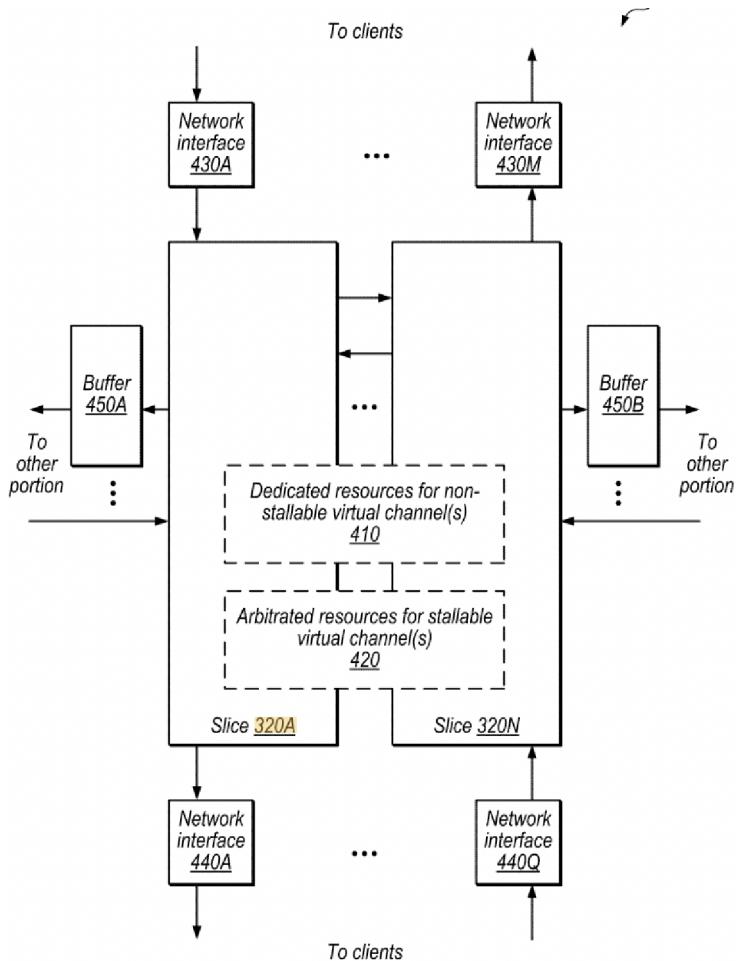
be fooled! The *tile* of the patent title does not refer to something like chiplets, rather it refers to something like a router. So the idea is that within a GPU core (and then core to core) there are various different clients (eg cache, datapath, and memory interface [which presumably means connection out to L2?]) all connected by a fabric.

This is the lower level of the GPU NoC; the higher level is what was described by the previous patent.

At the most abstract level, this fabric looks like a set of routers connected by wires, and that's hardly novel or patentable. So what's new?

There are two main interesting elements.

The first is that when Apple says “tile” what they seem to have in mind is something like an abstract template in their EDA tools. This template is a somewhat generic router looking something like



The idea here is

- the east-west routing (450A and 450B) communicates from one tile to another (so approximately from one core to another, though something like L2 probably also has its own tile)
- each north-south routing slice handles one client within a core

There is potential here to vary the number of clients (for example an L2 tile might have a different number of different clients from a core tile) and also to vary the number of east-west inputs or outputs. (Varying the number of inputs or outputs is probably for future designs; the current design connects all these tiles in a linear chain which more or less forces the same number of east-west inputs and outputs. The next step would be to upgrade the chain to a ring, and then to some sort of more sophisticated

topology.)

Each slice can potentially route to another slice on the same tile (ie communication within the same core) or to an adjacent tile (ie core-to-core communication or, more commonly, communication with the L2 or the companion core).

There's certainly potential for simultaneous communication within the same tile, eg two on-core elements communicating with each other at the same time that packets are forwarded from the east side of the tile to the west side; and the major portion of the patent is concerned with optimizing that. It describes a scheme that, every cycle, doesn't just arbitrate one potential route (ie one input packet is transferred to one output) but simultaneously arbitrates across all possible routes, to find the optimal subset that can occur simultaneously and have highest priority.

And of course it wouldn't be an Apple patent if there weren't some discussion of priority, which includes two elements

- first is that one priority class always gets to win (presumably control, interrupt, stuff like that)
- for the second, lower, priority class there's a scheme that tries to balance fairness across inputs over time.

You could imagine many fancier addition to this, the usual suspects like handling priority inversion, giving different inputs different weights, etc, but at least for now the idea seems to be to handle all that at a much higher level (per kernel, per threadblock, per warp) and just ensure that these tiles are approximately fair, while being able to handle maximal bandwidth every cycle.

The linear chain, as described in the patent, can't be completely correct; specifically there can't be a single tile representing "the L2 cache"; the bandwidth just wouldn't work – essentially such an L2 scheme could only service one core request per cycle, which seems far too low!

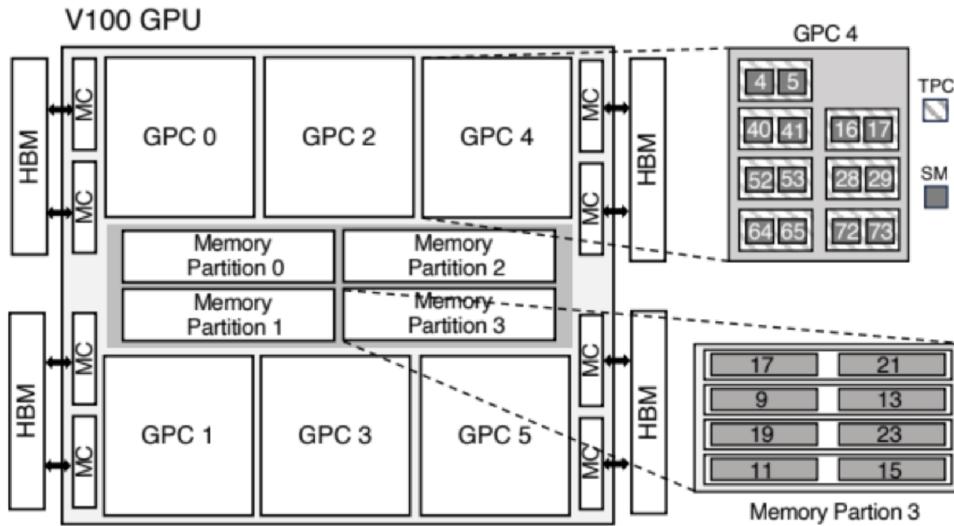
I'm guessing it's more like there's an L2 tile say every four or five tiles, so that L2 values can be injected fairly close to their target core, and multiple injections can occur in the same cycle. In principle you might also want more than one tile for, say, the companion core; but maybe the bandwidth requirements in that case are not so stringent as to demand more than one tile, even for say an M3 Max?

One way to think about this stuff is to compare it to nVidia. (2024) https://drive.google.com/file/d/1GK8Ypt_4V1LcABFtB5t_8Q4iGUse_Cia/view *Uncovering Real GPU NoC Characteristics: Implications on Interconnect Architecture* is a somewhat unsatisfactory paper, but has some interesting elements.

It describes nVidia designs as built of a hierarchy of

2 SMs in a TPC	40 TPC in V100,	54 in A100,	66 in H100		= core
(2 to 3 TPCs in a CPC)					= "A" ?
7 to 9 TPCs in a GPC					= "M" = ~8 .. 1
6 to 8 GPCs in a GPU					2 M' s = Pro, 4 M' s = Ma

For now ignore the left side of the table. We have something like:



So the GPU (eg V100) is built of GPCs which are built of TPCs.

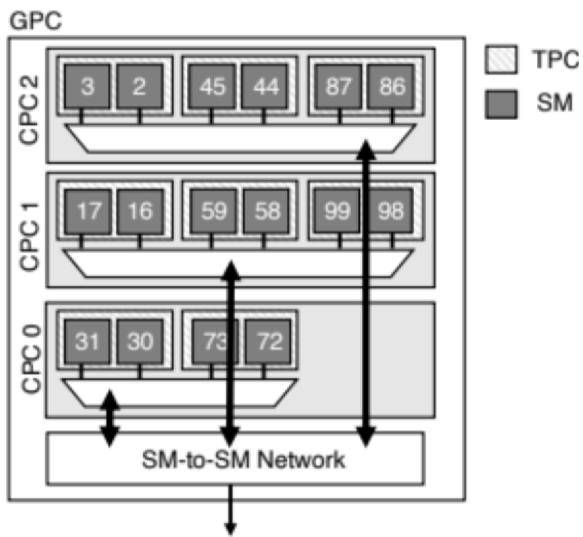
If we compare this to Apple we have essentially

- the TPC (Texture Processing Cluster) is more analogous to a core than the SM. As described elsewhere, An SM consists of four quadrants each 16 wide, and each instruction executes twice to cover the 32 lanes. So the TPC is the closest match to an Apple core, which consists of 4 quadrants, each 32 wide.
- The grouping of 7..9 TPCs in a GPC (Graphics Processing Cluster) is analogous to the 8 to 10 cores of an M CPU (eg M1, M2, ...) For lack of a better term, we will call this an M.
- Then the full nVidia GPU is made of multiple GPCs, in the same way that an M Pro is (?) made of 2 Ms, and an M Max is made of 4 M's.

One reason this analogy is suggestive is that nVidia presumably have physics dictating the structure of their hierarchy, and one suspects something like the same physics dictates the Apple hierarchy, implying that the GPC/M is a good compromise between items that need to work well together and the realities of limited communication radius at high frequencies.

If we move from Volta to the new Ampere and Hopper, we see two new features.

The first is that the TPCs are grouped into smaller units called CPCs (Compute Processing Cluster) which in turn are grouped into the GPC, as in



This suggests that perhaps Apple similarly (now or in the future) will perhaps group cores at say 5 or 6 together (call these an “A”), with two “A”s in an “M”.

The second innovation nVidia was forced to add to Ampere and Hopper was to split the entire GPU into two halves (“partitions”) which is obviously reminiscent of the Ultra split into two partitions. This nVidia investigation at least suggests some natural ways in which Apple might partition the GPU. The implications of such partitioning are that communication is lower energy and faster within a particular cluster level, and costs more if we have to transition to a different cluster.

(For the most maxed out nVidia you can buy as a consumer, the Blackwell numbers for RTX 5090 are 11GPCs consisting of 85 TPCs, so in our hand-waving analogy you can view this as something like the equivalent of 3 M Max’s [each Max is ~4GPCs], and the equivalent of about 85 GPU cores; for Hopper 4090 this was 11GPCs consisting of 64 TPCs.

Does this analogy work? Depends on what you’re doing.

If you’re comparing laptop with laptop, it works fairly well, with an M4 is about a quarter of a 4090 laptop, and an M4 Max about matches a 4090 laptop.

If you’re comparing desktop, then it’s better to imagine the SM as being the equivalent of a full core (helped by higher GHz, more L2, more memory bandwidth, etc) in which case a desktop 4090 is about twice an M4 Max.

The laptop 4060 is then at about the level of the M4 Pro.

Lots of handwaving here, extremely dependent on the exact details of what you’re doing; and in both cases there are intermediate purchase points, eg the M4 Pro with either 16 or 20 GPU cores, but this gives one a way to organize one’s thinking.

At least part of what’s going on is that as for M4 Apple can run two FP16s per cycle per lane, but not two FP32s. nVidia can run two FP32s per lane. This gives nVidia a theoretical 2x advantage in raw FP32 FLOPs; the extent to which that actually matters is one reason for the wide scatter in comparison results.)

Recap

At this point it's worth summarizing everything we have seen, both as a reminder, and to see where there are gaps in our understanding.

We start with multiple processes, each able to submit commands (ie kernels) which are tagged by a QoS via multiple command queues.

The value of multiple command queues is that within a command queue, commands have to be processed in-order, but no ordering exists between command queues.

Our first task is to choose the next kernel(s) to execute.

On the one hand, on a throughput device, the primary goal is to execute as many operations per cycle as possible, without too much concern for how long an individual item takes. This encourages the execution of one kernel at a time, for maximal cache reuse.

On the other hand, we would like, as far as possible, to always be executing code on every separate specialized piece of hardware on a GPU. This encourages the simultaneous execution of at least a few different kernels, ideally each mostly targeting a different piece of hardware.

Apple's solution involves multiple parts. The rough idea, though details remain unclear is

- selection of kernels by priority. At any time, the highest priority kernel is chosen for execution then, based on various factors, subsequent kernels, again based on priority.
- kernel priority consists of placement within a priority band. Processes that have been ill-served in the past have their priorities raised within a band, but can't move to a higher band.
- processes within a priority band are given target throughputs (ie fractions of the overall GPU during a specific time, like a frame interval) and are expected to match their workload to this target. Processes that exceed their target will have their GPU commands killed, and are expected to throttle back in response.
- specialized types of hardware include vertex processing, fragment processing, and ray tracing. Kernels are tagged by the compiler as to their use of this hardware, which will be used to try to ensure all hardware is being used simultaneously.

You can see that this scheduling is attempting to

- ensure that highest priority code (like UI) is not, and cannot be, blocked by lower priority code
- within a priority band, one app has limited ability to ruin the experience (take up all the GPU) for another app
- once we have ensured the above fairness rules, ensure that the hardware is used maximally

So we now have, at any one time, a “primary” kernel and a few secondary kernels. We continue the above goals by handing out threadblocks from the primary kernel, interleaved with threadblocks from the secondary kernels, but dialing back any secondary kernels that appear to be throttling (ie fighting for the same resources as) the primary kernel.

Running this primary+secondary kernels not only utilizes specialized hardware, it also

- allows code to execute even while a kernel is approaching its completion and can no longer fully occupy the GPU or even a given core. We have various technology (like duplicated registers) that allow us to begin the execution of a new kernel simultaneously with the winding down of an old kernel.
- between kernels (or more precisely, between kicks) we have to perform various L1 cache cleaning operations, and those can be executed simultaneously with newly executing kernels as long as the newly executing kernels are from independent queues.

There are (more aggressive) rules for synchronizing and flushing L2 data at Command Buffer boundaries (to synchronize with the CPU), but the details of this are unclear. One would hope that these are strand-like mechanisms (ie only processes that have opted into this expensive L2 flushing) have to be delayed by it, that simultaneously uninvolved kernels can continue execution, as opposed to a flush-and-block type synchronization that halts everything for everyone, but it's unclear.

So at this point what we have is essentially the companion GPU has a few designated active kernels, and its job is to hand out threadblocks until the kernel is done. That begins by defining the threadblock geometry (how many lanes of $x \times y \times z$). We've seen that the companion GPU attempts as far as possible to make blocks "square-ish" to maximize the possible internal overlap/reuse of data from one warp to another. The threadblocks are then handed out to cores as "batches", both to reduce bus traffic, and, again, to hand out "square-ish" units to maximize data reuse along internal edges.

The only control provided to the developer of threadgroups is that you can specify `[max_total_threads_per_threadgroup]`.

Is it worth providing API to allow the developer to specify

- threadgroup geometry (how many lanes of $x \times y \times z$)
- how threadgroups are handed out to cores (both the $x \times y \times z$ of chunks of threadgroups handed out to each core; and the order in which threadgroups are read from the kernel – perhaps there is value in starting at the center and spiralling outwards, or whatever?)

Certainly in the case where one is handling a single long array, one sometimes, as the developer, knows that the work at the beginning of the array is quick, and that at the end of the array is slow, and so there is value to looping over the array backwards, as a bin packing problem (ie you do all the slow work at the start, then as you progress each unit of work is fast and can easily be executed on any available core, and you have to wait a lot shorter for stragglers).

The actual handing out of threadblocks is done in multiple stages.

The first stage, executing on the companion core (one "designated" companion core on an ultra) defines batches of multiple threadblocks, which are (essentially) placed in a queue with markings as to how subsequent batches are tied together. That is, a given batch (and all successor batches, so the entire kernel) may be marked as executing on a single core, on a single chiplet, or across multiple chiplets. However this is a "virtual" marking; the target core or chiplet is not specified, only the point that these batches all bundle together.

The next stage, also on the designated companion core, converts the virtual target core or chiplet to a physical core or chiplet, based on power and current activity levels, and sends a batch to a specific

core.

The final stage, happening now on a GPU core, walks the batch, executing each threadblock in the batch.

Obviously this scheme has the potential, based on specific choices at each stage, to balance cache reuse against using as much hardware as is available, and against power. Also, obviously, it has the potential, should this be desired, to provide some sort of guarantees and hard caps on GPU usage, for example restricting a VM, and only that VM, to use 8 GPU cores. We have seen that nVidia has made use of this functionality to do things like allow a department to lease out fractions of a large GPU, and Apple could do similar things if there's a business demand. Apple's scheme could also be more general, in the sense that a VM might have guaranteed 8 GPU cores, but able to spill over to more if those are free.

One of the advantages of having much of this preliminary scheduling and QoS happen on a companion core is that the firmware can be rewritten and given new features with OS updates. In principle Apple could also even allow the user to supply "scheduling" functions for the first stage that allow the developer to define threadblock and batch geometry, and the order in which batches are extracted from the kernel's grid. The ultimate goal would be to decouple data layout (which might be an n -dimensional grid, a graph, or anything else) from thread execution layout (which might be tailored to the precise cache/communication geometry of a GPU, to allow for maximal lane communication between lanes, via L1, and then via L2) in the same way that nVidia has now achieved this...

We move down to a core, which is now tracking some number of threadblocks, and some number of warps within each threadblock. Our core consists of four shader "quadrants" along with some number (perhaps only one?) of specialized hardware like vertex, fragment, and ray. So obviously we will try to have active at any time warps associated with all of these specialized hardware blocks.

However if our concern is GPGPU, then the hardware blocks we care about are primarily datapath and load/store, the best we can say for the other blocks is that hopefully they can do their graphics thing and keep the GUI running smoothly while not much interacting/interfering with our code.

The execution of code as clauses means that, even for GPGPU code, we can fairly easily assume almost constant overlap of load/store and datapath execution, with some warps executing datapath clauses and others executing load/store clauses. Our goal, then, is to get even more simultaneous execution of every hardware block!

We avoid waiting on memory by the usual GPU mechanisms, and we avoid some waiting on branches (ie predicated execution) by Apple's speculative execution (begin predicate execution on all lanes, and only delay waiting for the predicate mask at the final stage of register write).

We avoid waiting for the instruction cache/TLB via clause L0 caches.

We avoid waiting for results by scheduling from multiple warps, trying to ensure that there's always a warp available with all its operands available. We use the multiple mini-front-end scheme to improve on this, though this scheme seems not yet fully optimized (doesn't seem to yet fully wait until all operands have been collected from the operand cache).

In the past we might be limited in how many warps or threadblocks we could schedule because of the

fixed pools of SRAM (for register storage and Scratchpad) but with M3 that's much less of a limiter.

So we can assume that much of the optimal scheduling machinery is available. How can we do better?

On nVidia, the load/store lanes are not as wide as datapath lanes, which means that a load/store instruction will execute for multiple cycles, during which other datapath instructions, possibly from other warps, can be scheduled. That's very different from how Apple handles this, via clauses, but gets to the same goal of aggressive overlap of load/store hardware with arithmetic hardware.

However the nVidia idea can be (and is) used by Apple for the Special Purpose Function hardware, so that one instruction, to say calculate a reciprocal estimate, executes over many cycles during which other independent instructions (from the same or other warps) may execute. The one tweak Apple makes to this is to try to ensure that we don't route too many instructions to this unit too rapidly, and therefore block. (As I've said before, many of these tweaks seems obvious on a CPU; the trick is how to implement them on a GPU without paying too much in area and energy.)

We could also substantially improve the performance of special functions (even user-calculated special functions like, eg Bessel) if we provide a nice set of "test" functions that can in one cycle test various special features of floating point inputs (zero? negative? infinity? NaN? denorm? etc) and allow code to run down a fast path vs the special cases path. These same "test" instructions could also be implemented for NEON and AMX...

With special functions out the way, what's left?

We sometimes need to shuffle data around. For some versions of this (eg RGBA rearrangement) we can get this done for free by dedicated hardware in load/store. We also have permute capability in the path from the Operand Cache to the arithmetic units. We do get free data shuffling as part of the internal execution of the matrix multiply instruction, but it would be nice to have every use of this permute capability (by the appropriate shuffle instructions) fused with the subsequent instruction so that the permute was, in a sense, free.

It would also be nice to have some API that explicitly allows using shuffle instructions as parallel lookup from a 32 (or 64?) element table...

We might also be able to use Texture hardware for the purposes of table lookup (and interpolation) and it might be worthwhile seeing what support could be provided to make this official, rather than a hack. At the very least Metal language support [eg language keywords that look like `const_table` rather than `texture`], and some control and guarantees over numeric details.

With shuffle hardware out the way, we are left with three primary execution units: int, FP16 and FP32. nVidia, when they went all in on 2-way dispatch, suggested that, as a rough approximation, about 30% of their datapath instructions correspond to uniform or int. So if we widened the mini-front-end and allowed two-way issue (probably from different warps, that's easier) to int and FP we'd get some overlap, though not a spectacular amount. If we added a 33rd lane to allow uniforms to execute separately from mainline datapath we'd get closer to that 30% (though Apple probably needs less integer math to calculate addresses than does nVidia).

So that's not bad – for the cost of 3% for an additional uniform lane, and some simple work on the mini-

front-end we get maybe a 20% boost in throughput.

We could also try to schedule FP16 warps simultaneously with FP32 warps. This might work better than one naively expects, in that it would allow GPGPU (I assume mostly FP32) to execute simultaneously with GUI (I assume mostly FP16).

We can be even more ambitious in a way that's probably still a net win given the extra hardware:

Apple have consistently separated FP32 from FP26 hardware to optimize each, and get FP16 graphics as low power as possible. But perhaps, at least for the Pro and Max chips, it's time to rethink that? Suppose we tweaked the FP32 just slightly to also calculate FP16.

Then we have at least two options

- we can preferentially execute FP16 instructions on FP16 hardware, but if an FP3 instruction is not available for scheduling, we can execute a second FP16 instruction on the FP32 hardware
- could add some FP16 vec2 instructions, just like AMD and nVidia, and execute these in the FP32 unit. This requires some more extensive work on the FP32 unit, but of course also gives us more throughput

So all in all we see there's still scope for Apple to improve the GPU (especially in the GPGPU ways I care about) without having to resort to raw doubling the number of cores.

There's also the option of dramatically changing the execution model, like nVidia did with Volta, to allow for non-lockstep lane execution, opening up a whole new set of algorithms and target use cases. Or the provision of specialized synchronization-type operations to speed various special (but common) use cases. I suggested, for example, gate instructions to allow execution of cleanup code by the first or last lane of a warp, warp of a threadblock, or threadblock of a kernel.

Data Cache Issues

After all this we can perhaps now appreciate more the implementation of some of the fancy cache ideas we have already seen in the SLC.

Some obvious features of graphics work compared to CPU work are

- there are rules for "software" cache synchronization, so basically forced cache flushing at various points.
- there's a lot of data streaming (ie 1-time use).

Ideally we don't waste cache lines on this data.

- there's a lot of temporary data, ie data generated in kernel N to be used in kernel $N + 1$, but then never used again.

Ideally after this is used by kernel $N + 1$ it can be flushed from all caches without ever being written back to DRAM

- there's a lot of data that is reused, but at a per-frame granularity, so that it's a long time (in terms of cycles) before reuse.

Ideally this can be both rapidly flushed from L1, maybe even L2, to allow those lines to be reused, BUT also retained in SLC.

Achieving all these goals is a constantly evolving set of ideas.

accelerating software coherence

Recall that the Metal cache coherence rules are

- between kicks (basically shader kernels) we have to flush the L1 cache, so every subsequent kernel sees whatever changes were made by the earlier kernel
- between command buffers we have to flush the L2 cache, so that the CPU can see whatever changes were made by this command buffer.

(2016) <https://patents.google.com/patent/US20180181491A1> *Targeted cache flushing* discusses how to accelerate this second task.

Conceptually each read or write of a line into L1 is tagged by the Command Buffer ID of the relevant instruction; that ID propagated down to each core as a special register along with other details like the threadBlockID and so on). This tag propagates out to L2.

Then at the end of a command buffer, an instruction can be sent to the L2 requesting that all lines tagged with the particular Command Buffer ID be flushed (out to SLC). This can happen in parallel with starting the next Command Buffer (if that Command Buffer is “independent enough”, eg associated with a different process), and is obviously more performant than pausing the GPU until the entire L2 is flushed.

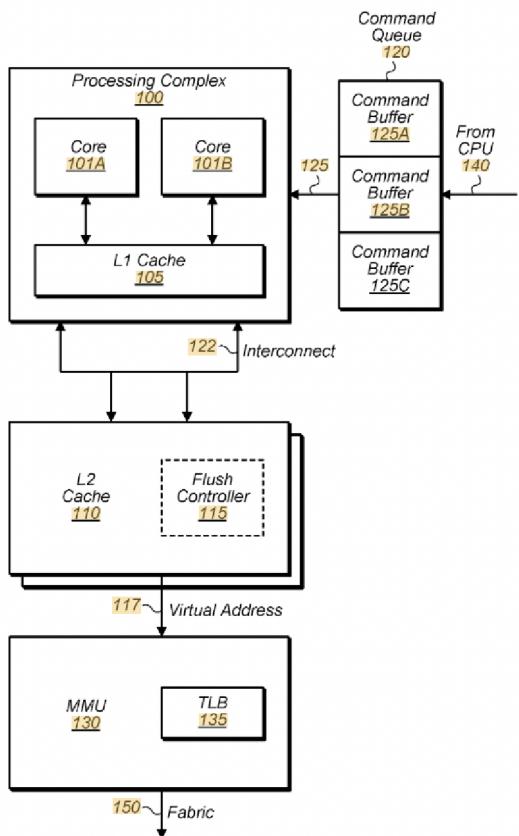
There are a variety of interesting issues mentioned as asides in this patent that are worth noting.

On the obvious side, the patent points that, in principle, we could also associate a kickID with each cache line in the L1, and use that to accelerate the flushing of L1 between kicks. But it’s not stated whether Apple does this (perhaps not at the time, but now?)

Less obvious, but unsurprising, is that you have to consider the possibility that two separate command buffers modified a cache line, eg one after the other. There is an additional “multiple ID” bit in the tag alongside the CommandBufferID, and this bit is set in the rare cases when this happens. Obviously every line with this bit set is flushed when a command buffer ends; cheaper than some complicated method for tracking exactly which of the multiple command buffers all touched the line.

TLB

But the real win in terms of unexpected understanding is this diagram!



So what you say? What's interesting here is what it tells us about the TLB! We do everything inside the GPU proper using virtual addresses, and only bother translating to a physical address at the point where the GPU has to interact with the rest of the SoC.

This is a huge energy and performance win. The state of the art in traditional GPUs (where constraints include having to work with an external CPU, and trying to present a unified address space over PCIe) is to operate the GPU much like a CPU. nVidia in particular operates the L1 caches in virtual address space, but the L2 cache in physical address space, so that translation has to happen for every access to L2. There's been a whole lot of work to try to limit the costs of this, but they are high.

some comparisons with nVidia's TLB and academic ideas

assuming the TLB is like a CPU TLB...

The obvious assumption is that we design a GPU TLB like a CPU TLB, meaning, for example, that the GPU L1 caches are *physically* addressed, and every load/store interaction with the L1 involves a TLB lookup. It turns out that this is a terrible idea, even though it was the background assumption to most of the early papers (which, honestly, makes them therefore irrelevant to anything!)

Let's start by looking at some of these early papers which give some relevant numbers.

(2014) <https://courses.grainger.illinois.edu/ECE598MS/fa2021/papers/paper233.pdf> *Architectural Support for Address Translation on GPUs* which gives some basic numbers; possibly somewhat different for pure graphics as opposed to GPGPU, but at least some insight.

Let's go through some points.

Obviously a unified address space is an important part of the Apple GPU design, which means we need something like a GPU MMU and TLB setup. But a CPU MMU performs three different tasks

- address mapping (and at that same time check/set various flags like permissions)
- taking unrecoverable faults (eg permissions, or accessing an unmapped page [eg NULL pointer])
- taking *recoverable* faults (eg page is marked compressed or on disk, so we have to switch to CPU to pull in the page, then restart the instruction)

Handling a recoverable fault requires

- a way to swap out the instruction until the fault is handled (that we mostly already support)
- + but a single warp could, in theory, generate 32 different loads all from different addresses, so this restart/swap out procedure has to be robust to somehow handling all 32 loads (and if we handle them sequentially, load/fault/swap, load/fault/swap) we need to ensure that the 1st page hasn't been swapped back out by the time we get to the 32nd page...
- some sort of way to restart the instruction (that's something of a big change)

- some OS software capable to doing the work of handling the page fault. Do we want to do this on the Companion core? That's surely faster, but means the Companion core has to know about the File System! That's probably too much to demand, at least at first.

So now we have to have a way to propagate the fault request to another core. (Though we could imagine new options, like making the Companion core a full E-core, running at least some subset of the full OS. Maybe there could be advantage to this, especially if we start giving enough extra work to the Companion core, like user-managed scheduling?)

OK, so we've established one important possible difference between the GPU MMU and the CPU MMU; but primarily we are interested not in faults but in address mapping. So let's consider some numbers. For now we consider just load/store.

The conventional wisdom in GPU coding is that about a quarter of instructions are load/store, and this paper confirms that.

Recall that while nVidia interleaves load/store instructions with datapath instructions, our belief is that Apple's clause-based system means that warps will run clauses of pure load instructions, clauses of

pure store instructions, and clauses of compute instructions. This has multiple nice side effects (like allowing us to simply track when an entire load clause is complete before starting a subsequent compute clause) but let's think what this means for implementation.

Apple, in their patents, talk about load clauses and store clauses as distinct, and this kinda makes sense – the two have different implications in terms of their interaction with other clauses. But even if the clause types are distinct, they could be executed on the same hardware. Or on different hardware. Who knows? Let's assume it's mostly the same hardware, with "load vs store mode" switching a few control signals; things don't change too much if it's separate hardware.

We only need load/store to operate at a quarter of the rate of the datapath. One way to do this (given four datapath blocks, each 32-wide) is four load/store blocks each 8-wide, executing each instruction over four cycles. But what value is there in associating a load/store block with a datapath block if there is a common pool of registers, not a per-quadrant pool (as seems to be the case, at least as of the M3)? If we look at things that way, maybe it's better to have a single load/store block 32-wide working in conjunction with four datapath blocks also 32-wide?

Either way, we are in the position that, approximately, we want to serve about 32 load/stores per cycle. Which, before we get to issues of how to handle cache access, means TLB access. Obviously these are terrible numbers – 32 TLB accesses per second! Which explains why nVidia don't even bother, translating only with the less intense stream of (coalesced, line at a time) requests from each L1 out to L2. The numbers in the paper, no matter how they try to sugarcoat it, show that a traditional TLB sitting before the L1 cache is probably not going to work.

The takeaway from the above should be that

- placing the TLB in front of the L1 has terrible consequences (for area, for power, for latency). Even if you use tricks like consolidating multiple load/store requests to the same page, it still looks bad, very bad.
- doing things this way also seems likely to complicate any attempt at supporting recoverable faults, if we should one day want these.

We can get some possible further insight by looking at (2017) <https://tbennun.github.io/papers/gputlb-damon17.pdf> *Big Data causing Big (TLB) Problems: Taming Random Memory Accesses on the GPU*, which describes the TLB layout of nVidia chips at the time.

This paper suggests that Kepler (2012) has

- 16 L1 TLB entries, each covering a 128kB page
- ~65 L2 TLB entries, each covering a 2MB page
- ~1032 L3 TLB entries, each covering a 2MB page

and Pascal (2016) has

- 16 L1 TLB entries, each covering a 2MB page
- ~65 L2 TLB entries, each covering a 32MB page
- looks like no L3 TLB

At least as of Turing and Volta (I can't find values for Hopper) we have what looks like the same as Pascal except four times as many L2 TLB entries

All this suggests two things

- there's real value in having large pages available. The next best thing (but not as good) is having a large TLB available.
- why exactly do you *have to* have your L1 cache operate in physical address space? On the CPU this is done because of various historical assumptions about how code might possibly map the same physical page into multiple virtual pages for various (generally unlikely and uncommon) reasons. But we don't have to make the mistakes of the CPU on the GPU. By taking advantage of this, nVidia can dramatically reduce the number of lookups required, only for lookups that miss in the L1 cache. Also by doing things this way, if nV want to handle faults, this is now out of line relative to instruction execution; if the TLB wants to generate a fault that's somehow propagated to the OS and results in a disk access or whatever, that's quite feasible, it just looks like a very slow lookup in the L2 cache.

To cut a long story short, Apple, faced with the same problems and constraints

- operate both their L1 and L2 caches in virtual address space. (In A17/M3 and later, L1 gets more complicated as we will eventually see below). This means the Apple GPU TLB and MMU sit at the exit of the GPU, between the GPU (operating in virtual address space) and the rest of the SoC (operating in physical address space). This further reduces the bandwidth required by the TLB, and allows it to operate at slightly longer latencies (eg to be more complex, or to be larger).
- Apple is constrained to the use of 16kB pages. That will probably change at some point (we have seen multiple hints of support for new page sizes in various patents). But for now the solutions Apple seems to have adopted to limit the problem are (all described later in much more detail)
 - + use coalesced pages (so that 2 or 4 or 8 sequential virtual pages entries with identical permissions and sequential physical addresses are handled by a single entry in the TLB along with some per-page validity bits)
 - + use a large TLB
 - + provide a sophisticated MMU cache alongside the TLB, so that even when individual pages miss in the TLB, the lookup of the new page (starting at the page table root) normally hits intermediate table nodes cached in the MMU
 - + use the DSID mechanism to cache page table entries in the SLC.

How does nVidia handle the bandwidth?

For Kepler the L1 is private to one core, the L2 is shared by *three* cores, and the L3 services the entire GPU.

For Pascal, the L1 is shared by *two* cores, the L2 is shared by *ten* cores.

(Note that these mean that the L1 TLB sits at the exit from one or two cores, placed between those two cores and access to the L2 cache. My guess is that the rules are only physical addresses on the L1↔L2 bus, so the translation has to happen essentially within each core. An alternative might be to send virtual addresses to the L2, but the nVidia scheme allows for distributed translation, so higher bandwidth.)

Overall, I think it's fair to say that TLBs in the GPU, especially in systems where the CPU and GPU are tightly coupled, can probably be further optimized. A big problem is that GPUs use memory very

differently from CPUs.

One type of use case is streaming, and this can perhaps be handled by prefetching in the GPU (either data prefetching or just TLB prefetching).

But a different type of use case is very large datasets with somewhat random access patterns, and the most obvious solution for that appears to be *large pages* along with *very large TLBs*.

Large pages means technology to have smaller pages (optimal for the CPU) working alongside large pages, and this is basically a well-understood problem even if Apple has not yet done anything visible in this space.

Regarding large TLBs, an idea that may ultimately win out is to use the LLC to also store TLB entries. The LLC is, of course, huge compared to even thousand-entry TLBs, and with some appropriate control flags to ensure entries are locked into the LLC appropriately (“long enough, but not too long”) this might work well. The main technical difficulty is the lookup, given how LLC entries are look up up by physical address, whereas TLB entries are looked up by a combination of virtual address and ASID. You can imagine ways to solve that, one obvious suggestion is to hash the virtual address+ASID into something that looks like a physical address, along with one extra “TLB vs data” bit. A slightly different solution is suggested in (2019) <https://dl.acm.org/doi/pdf/10.1145/3309710> *DUCATI: High-performance Address Translation by Extending TLB Reach of GPU-accelerated Systems*, written by some folks at nVidia. Of course you then also need some machinery to track down and invalidate these LLC entries when TLB mappings are invalidated or changed.

As I said, Apple seem to be doing this via the DSID mechanism (to be explained) which gives most of the performance benefits of something like DUCATI while avoiding the problems resulting from invalidation.

Apple's MMU/TLB solution

We've seen that Apple can avoid some of the nVidia's issues by placing the TLB at the “exit” of the GPU L2 cache, but we still need process isolation. How is that handled?

This gets us back to the mysterious “Contexts” discussed much earlier in this document. Rather than translating every virtual address into a physical address, conceptually every virtual address has a 4-bit contextID prepended to it so that effectively it's a 68 rather than 64 bit address. This contextID is prepended to the address by the load/store unit and propagates through L1 and L2, preventing what is apparently the same virtual address (same 64 bits) from colliding if the two addresses come from different contexts. Right now there appear to be 16 contexts (ie 4 bits) though of course that could easily grow if required.

The MMU then uses the contextID somewhat like an ASID to translate addresses from virtual space to SoC physical space.

The MMU is not just a TLB, it's also a proper MMU cache, so it appears to have, according <https://github.com/AsahiLinux/docs/wiki/HW%3AAGX> a structure like

- L0: 2 entries
- L1: 8 entries
- L2: 2048 entries

- L3: 2048 entries

Don't be fooled! This does not mean something like L0, L1 and L2 TLBs!

Instead it means that we have dedicated storage for holding that many entries to walk different parts of the page table tree.

Recall that we translate a page by doing something like splitting the page number into (say) 3×11 bits, and using the first 11 bits as an index into some table T0 to read a base address T1, then the next 11 bits as an index into table T1 to read a base address T2, then the final 11 bits as an index into T2 to read the physical address.

Table L3 caches physical addresses, so it's the equivalent of a TLB.

Table L2 caches the T2-type entries, likewise for table L1. These are the intermediate lookup points of the table walk and caching them dramatically speeds up a table walk by allowing us to short circuit a few of the early steps.

Table L0 essentially gets you started in the table walk, so will vary per context. Although Asahi says there are two entries I don't think that's quite accurate; I suspect it's more like there is one entry for OS use and one entry per context.

However even with all this to help us, 2048 entries each of 16KB only covers 32MB. That's only one of nVidia's L2 TLB pages!

Apple is helped by having the MMU cache so that a miss in L3 and so a table walk probably hits in L2, and we only have to load one page table entry (possibly from SLC, though more likely from DRAM).

The second thing Apple appears to do is use a coalesced TLB. When you load a page table entry from DRAM, you're probably loading a 128B line (maybe 64B, it's unclear, but the unit of transport around the SoC appears to be 128B lines). This means you're actually loading 16 successive page table translations. Now, if the permissions for these successive 8 pages are all identical, AND if the physical addresses are allocated by the OS to be linearly increasing, then in some sense a single TLB entry can correspond to these 16 pages. You can do something like store the data for the first page of the sixteen, along with 16 validity bits [or some other indicator of how many pages this "superpage" covers]). We've discussed coalesced TLBs of this sort, with some academic papers, in the earlier CPU section.

The relevance of this is that there's a patent, *Multi-block Cache Fetch Techniques*, which suggests something like this is going on, that a single tag (ie entry) in the TLB has multiple associated validity bits so as to cover multiple pages. Below we'll discuss this patent in a lot more detail.

Increasing the effective size of the TLB by a factor of ~16 say is still far below nVidia's TLB reach (about 8GB? maybe more on the largest new GPUs?) but it gets us to half a GB which is a start, until Apple SoCs support large pages. And on the positive side, when Apple does miss in the TLB, the cost of the page walk is usually quite a bit cheaper than for nVidia.

DSID and dropping cache lines

We've seen that tagging some lines in L2 (and possibly L1) help us with more efficient flushing of caches to maintain software coherence at Command Buffer (and possibly kick) boundaries. Can we extend that idea?

The target now is that much generated pixel data is temporary, only used for subsequent render passes, or as partial window content that will be composited by the Window Manager; but discarded after use. To utilize this fact, conceptually each read or write of a cache line is now also tagged with a Data Set ID (now in addition to the Command Buffer ID tag). This tag propagates out to L2 and SLC. The immediate point is that this means that lines in SLC are now tagged by a Data Set ID, and we can do multiple things based on that tag. In particular, for this sort of temporary data, once it's no longer required, we can send a request to SLC to drop all the lines tagged with a particular DSID. Lines associated with a DSID may be write-lines (in which case we don't need the data ever again, so can just toss them) or read lines (where we may read the data again, but much later in time).

This manual line dropping

- allows us to immediately reuse these cache lines for new data,
- while also not having to pay the energy and bandwidth costs of writing modified line data to DRAM

In addition, now that lines are tagged by a DSID, they can be used for the various other SLC smart cache functionality like quotas for, or locking lines with, a particular data set ID.

The hardware side of this is described in (2017) <https://patents.google.com/patent/US20180349291A1>
Cache drop feature to increase memory bandwidth and save power.

This is implemented in SLC by having attached to the side of the SLC

- a DSID Mapping Table, which holds information about various DSIDs
- a Drop Control Unit, which can autonomously walk through the cache one line after another, invalidating each line that matches the DSID(s) that have been selected to be dropped.

The Mapping table looks like

Virtual Data Set Identifier	Physical Data Set Identifier	Current Number of Cache Lines	Quota Number of Cache Lines	Drop in Progress
58 (N/A)	0	0	42,000	No (N/A)
80	1	28,843	50,000	Yes
81	2	15,823	50,000	Yes
82	3	32,401	50,000	Yes
.				
121	28	8,993	33,000	No
125	29	20,982	45,000	No
254 (N/A)	30	0	38,000	No (N/A)
255 (N/A)	31	0	38,000	No (N/A)

There are a few non-obvious points here.

- To give approximate numbers (possibly increased since 2017) the system allows for 256 (actually 246) possible “virtual” DSIDs within the GPU (ie 8-bit DSID) but only 32 “physical” DSIDs within the SLC, so the table remaps the GPU values to the SLC values. This saves a few bits in the SLC but is no big deal either way. More interesting is that this same table stores the quota provided for each DSID and so is the point at which quotas are enforced. A later patent suggests that the number of virtual DSIDs has now grown to $\sim 2^{16}$.
- Non-GPU clients (eg camera or media encoder/decoder) can also use DSIDs, and exploit the same functionality like locked lines, quotas, or dropping all lines associated with a particular DSID.
- ten virtual DSIDs are unavailable to the GPU or other client because they are used by the SLC to tag specific types of data which then gets treated differently.

One DSID is what you might think of as “the CPU DSID”. It tags lines that come from the CPU or any other “traditional” client and that are subject to traditional LRU cache control. (This DSID might have been split in the M3 or a later design given that Apple is now treating some CPU cache lines, eg instruction cache lines, as higher priority than standard CPU data cache lines...)

Another DSID is for “compressed metadata” for whose meaning your guess is as good as mine!
Eventual compressed lines in the SLC?

Yet another DSID is for storing memory management unit (MMU) parameters and page tables, which might be a way of giving those extra “persistence” in the SLC, in other words something like a

way to achieve the goals of DUCATI (as discussed above in the TLB section, namely expand the TLB into the last level cache) in a way that's minimally disruptive because it mostly reuses existing technology (the entries are stored in the SLC looking like standard page table lines, accessible by the existing page walker machinery; but they can be made sticky in the SLC by using the DSID machinery).

It's probable, based on other patents, that at least one of the DSIDs is also used for hardware debugging, to tag specific trace data (eg sequences of memory transactions) which are stored in SLC until later read out.

What do we do if we cannot find a free physical DSID? In that case the data is treated as "CPU" data and given the default DSID. This means it isn't subject to all the various controls (quota, dropping, etc) and so will be less performant and will use more energy, but the system will keep working as expected. Of course we try to size things like the number of physical DSIDs to make this a rare occurrence.

Another nice tweak recognizes the fact that, of course, the scan to drop all cache lines with a particular DSID takes time. Suppose we need to allocate a new cache line during that time. Of the lines in the set of interest, if any way has a tag corresponding to a "being-dropped" DSID then that way will be treated as invalid and preferentially replaced, even though the walker looking for DSID lines hasn't yet reached it.

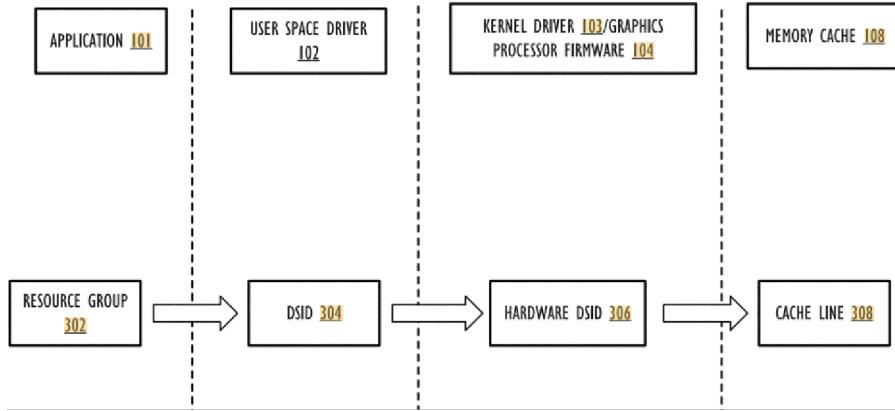
There are also a bunch of technical rules for the (presumably rare) cases when two DSIDs interact with the same cache line (ie same cache address in physical address space) in various ways, eg DSID1 writes the line then DSID2 reads the line. Mostly these rules boil down to "if this happens you're using DSIDs incorrectly, but things won't break, we'll just mark the line as having the 'CPU DSID' so that from now on it's governed by normal LRU cache rules".

One final issue that is not discussed in this patent (or anywhere else) is suppose the GPU emits a DROP command for a particular DSID. We know that the lines associated with this DSID in the SLC are dropped, but what about the lines associated with this DSID in the L1 or L2 caches? Obviously dropping them also makes sense, and could be performed by essentially the same hardware that is already scanning the L2 cache looking for lines tagged with a particular Command Buffer ID to be flushed to SLC, but I don't see this ever pointed out anywhere.

DSID refinement

A year later we see the software and implementation aspects of the above ideas. We have been somewhat cavalier in asserting that "somehow" a DSID gets associated with cache lines as they are accessed. How exactly does this happen?

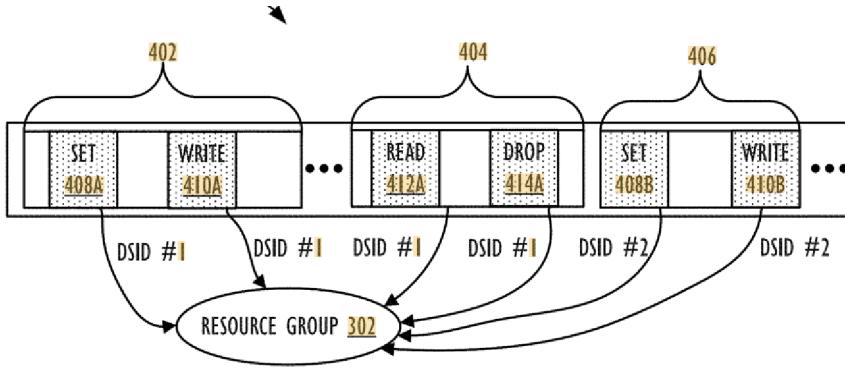
We see some details in (2018) <https://patents.google.com/patent/US10672099B2> *Memory cache management for graphics processing*. The diagram below shows the essentials points:



The application defines resource groups. (More realistically these are defined by the graphics APIs based on how resources are used by a chain of commands. Consider for example an ephemeral texture that needs to persist longer than one kernel, so it cannot be stored purely in tile memory, but which is uninteresting after being used by the second kernel. This is a resource with specific semantics, namely that it needs only “temporary” persistence and no “long term” persistence. Other resources can be grouped similarly based on the required semantics and expected uses, and thus their optimal flow through caches.)

Resource groups are then, within the compiler, associated with an abstract DSID, which is mapped by firmware to a concrete (not yet in use) DSID.

Meanwhile in each command buffer, one of the very first commands is a SET command which sets (presumably in some register in the GPU core) the DSID to use going forward for this set of commands. From that point on, until a reset, that DSID is used implicitly for all cache activity. Cache lines read into L1 (and propagated up to L2 and SLC) are marked with that DSID, likewise for cache lines that are written. Along with those traditional operations, there's also a DROP operation that can be used to invalidate all lines marked with that DSID as no longer interesting.



One of the many things that remains unclear to me is the extent to which workgroups from different command buffers can be interleaved. To the extent that this is possible, obviously the DSID register would need to be swapped, like the general purpose GPU registers are swapped, when these different workgroups swap control of the GPU.

If the above is possible, then it's possible (though the patent is somewhat vague on this) that this mechanism could also be used to accelerate certain types of fences and barriers. If workgroups from multiple command buffers can be interleaved, then a fence could use the DSID to identify reads and writes associated with the particular command buffers of interest, and enforce their ordering while not bothering with the reads and writes of other command buffers.

It's also interesting to realize that a similar idea is, in principle possible on the CPU side.

A standard CPU pattern is that a buffer is allocated (in virtual space, via something like malloc) on top of some pages in physical space. When the buffer is relinquished (via something like free) the physical pages are not released, and the (no longer valid) contents of the buffer will still write successively through L2 and SLC all the way back to DRAM even though no-one cares about them anymore.

We can limit the damage to some extent by trying to ensure that new allocations reuse the same virtual address range as that just freed, and thus the same physical pages, but there are limits to how well this can work.

One possible solution (hardly perfect, but better than nothing) could be to associate something like a TLB with each cache. The equivalent of a free would execute a cache_free(virtual_address_range) operation which would use the TLB to convert the address range into a collection of physical addresses which would be placed in the L1 free_TLB. Then the relevant lines in L1 would be limited from being written back. This could be done in various ways. We could wait until a line is chosen for writeback and test it in the free_TLB. And/or on occasions when the entire L1 is written out (eg when the core is put to sleep) we could match each line against the free_TLB. Likewise for L2 and SLC. It's a matter of choice the granularity at which we operate the free_TLB; page granularity is an easy option though obviously this means there will be some writebacks that aren't caught (basically any part of a free that does not cover an entire page). And of course we invalidate entries in the free-TLB once the address range is reallocated by a malloc equivalent.

Another year later we see a further evolution.

Suppose that during the construction of a frame we used some resources, eg material properties, or whatever. After we are done with these, we don't necessarily want to invalidate/drop the lines holding this data because we may need these resources in the next frame. But we don't have a strong preference for keeping them around because chances are there will be more important data to use those lines, certainly in the L1, probably in the L2, maybe, maybe not in the SLC.

(2020) <https://patents.google.com/patent/US20210096994A1> *De-Prioritization Supporting Frame Buffer Caching* implements this idea. The previous DSID scheme is augmented with a DE-PRIORITIZE command. This marks the data in SLC not to be completely dropped, but tagged as low priority. If

nothing more important comes along, it's available for the next frame; otherwise we'll have to reload it from DRAM.

Another way this functionality could be used is for buffers that have been written to (but that we want persisted).

Such cache lines now have a hint that the cache line will not be written to or read from again, so as soon as it's convenient the cache line contents should be moved up to DRAM and the cache line reused. We get some of the functionality of DROP in that the cache line is immediately available for use by another client, but this time also applicable to data that does need to be written out to a permanent location (like the screen).

The patent specifically suggests using this flag when building the frame buffer (and gives all sorts of complicated details I don't understand about how the flag interacts with other aspects of how the frame buffer is cached), but it seems to me that it's also more generally useful in the ways I suggest.

It's clear that the current DSID scheme is more flexible than the earlier Command Buffer ID scheme. On the other hand it definitely still feels suboptimal, forcing multiple resources all accessed within the same command buffer to be treated in the same way. One possible future solution might be to take advantage of the separate address spaces used by the GPU (as described earlier, eg a separate space for Scratchpad storage). One could imagine something like

- although the GPU acting on behalf of some process is sharing the address space of the process, give it a graphics-specific carve-out of that address space identified by highest 2 virtual address bits being 11 or whatever
- allocations by the GPU into that "secondary" address spaces follow GPU rules, not CPU rules
- when the GPU creates new resources in this secondary address space, it could allow the top 12 (or 16 or whatever) bits of the address to act as a resource ID. In other words we don't have to clumsily fake a resource ID by setting a DSID register that implicitly tags all subsequent memory addresses; rather we burn the resource ID into the address of the resource and then all subsequent accesses to that resource can extract the resource ID from the virtual address of the access at the same time that we access the TLB, and then stick this resource ID onto the relevant cache line.
- this would provide an ID that could propagate all the way to the cache like the current DSID but at a finer, per-resource, granularity without the coarse granularity of the DSID mechanism
- this new, fine-grained ID could then be used as the basis for hints to drop or flush cache lines, or raise/lower their priority, or use them as quota groups, or given them special cache behavior (LRU, MRU, locked, random replace, etc) at a much finer granularity than is feasible with DSID, and in many more situations.

There also seems to be scope for linking the DSID scheme with the TLB, ie applying all the machinery of DSIDs and manually managed caching to the TLBs. As far as I can tell all the work that goes into DSIDs and trying to ensure long term (frame-to-frame) persistence and reuse of GPU data assets is not applied to the page numbers (ie TLB data) of these assets. Imagine

- attaching a DSID to TLB entries and then

- + using that DSID to flush the TLBs when DSIDs are flushed, likewise
 - + (perhaps) providing an L3/SLC cache for appropriately DSID-tagged TLB entries, so they likewise can be persisted from frame to frame.
- There may also be scope for TLB-prefetching. Even if the GPU does not want to engage in data prefetching (too much risk of energy wasted on data that might not be used) prefetching TLB entries is lower risk and lower energy, with a high reward?

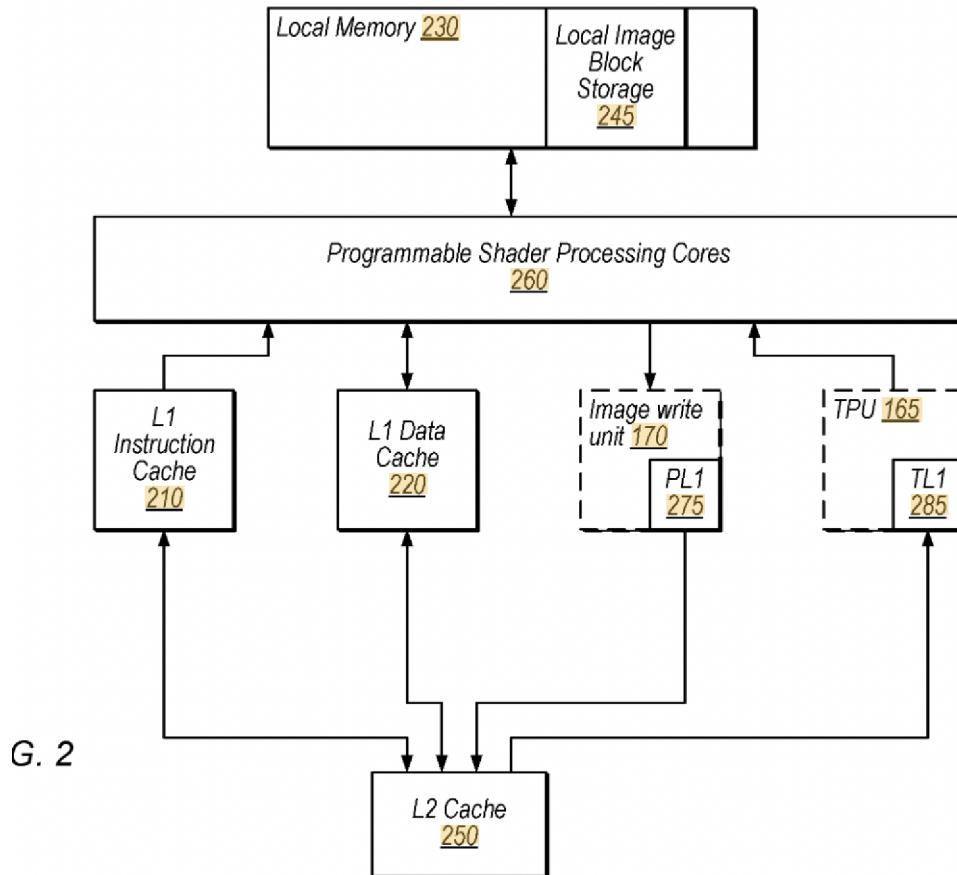
general data cache issues

GPUs have multiple address spaces and multiple special purpose caches. We won't cover all of these but to give the idea: as we have already mentioned

- each core has its own dedicated L1D cache (non coherent with the L1D's of the other cores)
- each core also has a texture cache (described in more detail when we get to the Texture issue section below) which is a read-only cache, and hooked into the texture decompression and texture address space scheme
- each core also has what's called an Image Write Cache. I *think* this plays a role in transferring data from Tile storage to global storage, once the construction of a particular Tile is over. But it's not at clear! In diagrams this cache is called PL1 which I think means Pixel L1, just like TL1 means Texture L1.

The point of the picture below is that there are at least four paths from L2 (shared) to various L1 (per core) caches. (On the M3 and later, to add to the diagram below, there's probably at least also a Ray L1.) This means lots of opportunities for different data in different caches to become incoherent, and some care needed to ensure that everything stays in sync according to the system rules.

On M3 this becomes simpler because these all share the same pool of SRAM, but different instructions are used to read/write that SRAM).



Some details are discussed in (2016) <https://patents.google.com/patent/US10324844B2> *Memory consistency in graphics memory hierarchy with relaxed ordering*, but I don't want to get lost in the weeds so I'm only going to mention the parts I consider interesting ways to improve the issues of cache coherency that I described in the introduction. These include

- provision of a Local Address Space that is per-core, along with a global address space shared across cores. This is Apple's term for what I have been calling Scratchpad. We can think of the Local Address Space as something like a range of addresses (let's say 64kB per core) that cover an SRAM that's attached to each core, but with the proviso that addresses in this range, automatically stored in this SRAM, *cannot* be written back to the L2. Scratchpad storage is not visible to another core, or the CPU,

or even the next threadblock that runs on this core.

So even at the end of a kick, there is no need to worry about writing the contents back to L2.

- very much like Local Address Space (and sharing the same SRAM) is Local Image Block Storage. This is essentially local storage with a defined width, height, number of channels, and bytes/channel. Thus it can be used as a small subsection of an image with many engage in address calculations handled by the system. Like Scratchpad, this Tile storage is invisible to anyone else; the system just needs to ensure at the end of a Tile that it is flushed out to L2 to build up the entire image.

- similarly there is a Global Address Space which automatically writes to L2 bypassing L1, which gets you automatic sharing (but of course you don't want to use this address space unless you have to because access will be slower; and just being shared doesn't magically give you ordering, you still need barriers or fences to ensure that the cores writing to and reading from the shared space do their work in the correct order).

- apart from these two somewhat obvious address spaces is the more interesting fact that some instructions provide a few precise cache control bits that can specify things like whether a load should be written out to L1 *and* L2 (so will be reused by this core, and also needs to be visible at some point to other cores) vs written out only to L2 (needs to be visible at some point to other cores, but will not be reused by this core soon).

There's also a *persistence* field in some instructions that indicates that stored data is expected to be ephemeral (or streaming) so that while it is being stored in L1 or L2 for temporary staging (reading values one time, or writing out a stream of data) it's unlikely to be reused and should be the first data to be flushed when a new line needs to be allocated.

Another way to look at these different fields is that

- sometimes we want stored data to be treated as cached *local* data,
- sometimes we want it to be treated as cached *global* data, in which case it needs to move to L2 more rapidly, and we need to be very careful about keeping a copy of this same data in L1.
- sometimes we want it to be treated as *buffered global* data (keep the data in L1 or L2 for as long as possible, allowing us to build up longer runs of modified bytes, and to include data overwrites, before the expense of transferring the data up to a higher level; but ultimately owned by the higher level)

These “buffered” lines are specially marked, and on various occasions (some sync instructions, end of kicks, etc) they are treated as buffers to be flushed to their real target.

(All this is not the same thing as DSID control.

DSID control allows for

- dropping multiple data lines once a resource is no longer useful
- control of large memory resources [high or low priority, locked or unlocked, quotas, etc] primarily in SLC

The persistence field described in this patent is about fine-grained control of especially the L1 and to a lesser extent the L2 cache.

It's almost a cache analog of the register operand cache control bits in most of the GPU instructions.)

Yet another set of control flags specifies whether the operations have acquire/release semantics. This has to do with how the stores or reads are ordered relative to prior stores or reads. We discussed this in the Memory chapter; recall that the essential point is that if a flag is meant to convey to another processor that a set of data is now ready to be shared, then all stores prior to the flag must be completed (or, more precisely, must be visible to other processors) before the store of the flag is visible, and similar constraints on the read side.

Of course you want most loads and stores (even those of shared content) to have the most relaxed semantics possible, limiting acquire and release purely to "flag" type loads and stores.

Finally control flags can modify the above attributes to clarify whether their ordering scope is limited.

Scopes can include

- within the lane (ie all we care about is the ordering of loads and stores as seen by this thread). This might seem bizarre, but there was enough out of order work, even back at the time of this patent that this could become an issue. Imagine, for example, a thread *constructs* a texture which it then immediately wants to *use* as a texture. We now need an appropriate barrier to ensure that the constructed texture is appropriately pushed out (maybe from PL1?) to a point of coherence (probably L2?) before the texture unit tries to read from that texture (placing it in TL1)...
- across a threadgroup (ie the set of lanes within a single core). This only requires flushing out from within per-lane storage (operand cache and write buffers and suchlike) to L1 (which is shared across lanes). There are also some atomic operations which synchronize across lanes and which may be more lightweight than using memory fences.
- across a device (ie the set of GPU cores), which requires flushing to L2.

It's easy to get lost in the details of all this. Remember that the overall goal is that we want to have most execution (including specifically loads, stores, and cache execution) as fast and low energy as possible, so with no overhead of enforcing ordering, snooping, or suchlike. Which in turn means we need special operations to *enforce* ordering, or cross-cache communication, when that is required.

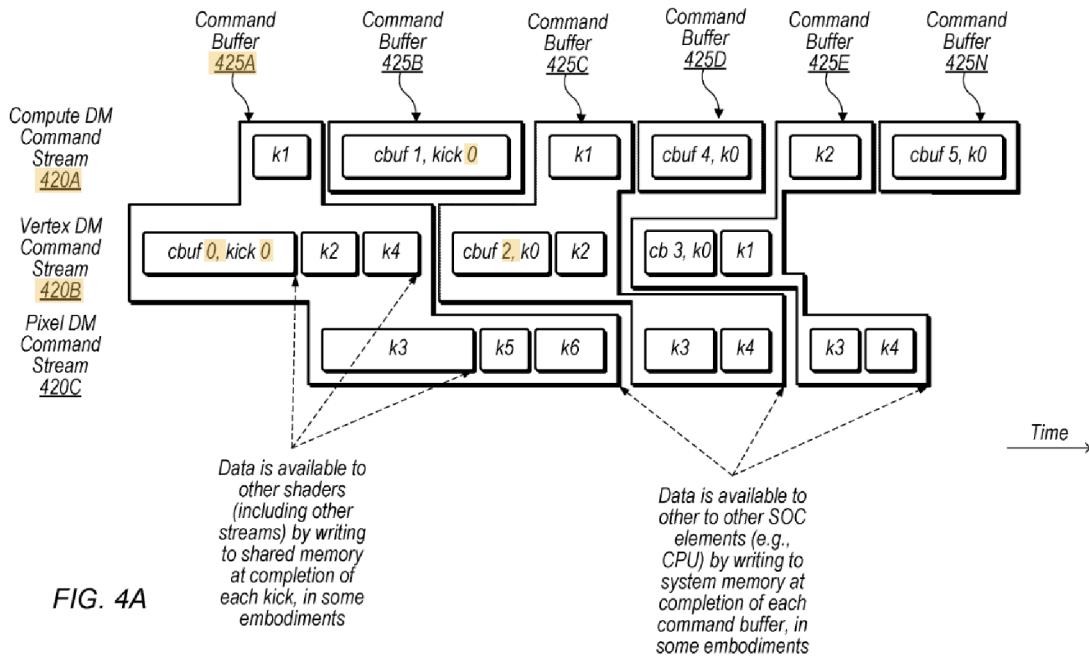
- By making those operations as fine-grained as possible, we can ensure that the bare minimum of work (ordering, and data movement between caches) occurs to enforce the specific synchronization required by the code.
- By providing these operations at fine granularity (eg not just as cache flush operations, but also as special load/store operations) you can avoid a whole lot of overhead.

In particular suppose that you want to follow up some graphics work with some compute work. In the minimal synchronization model we first described, this might require flushing all the L1D's to L2, so that the compute work sees the results of the earlier GPU work. But that flushing means possibly also

moving a whole lot of data that's not relevant to the compute work! If, instead, we can use these special load/store instructions to write out to L2 and then later read in to L1 only the shared data exactly required by the compute pass we can transition from graphics to compute at much lower overhead.

At this point we can revisit a diagram we have seen before, now with a fuller understanding of all the various parts.

The diagram below shows what this looks like from the software side.



GPU instructions are bundled into *Kicks* which are bundled into *Command Buffers*. The important points about these are that

- Kicks automatically (ie the compiler/GPU firmware will do the required work without developer involvement) synchronize data *within the GPU*, and
- Command Buffers automatically synchronize data *within the SoC*. (Note that Kicks and Command

Buffers are much larger concepts than Clauses. Whereas a Clause is like a CPU Trace, just a few instructions long, a Kick is something like a large function, and a Command Buffer is something like an entire program).

If the only primitives available were to flush the L1Ds to L2 this would be the end of the story. But the new primitives that we have introduced allowing specific data (not the entire L1D) to be placed in L2 mean that (possibly sometimes with developer assistance, possibly sometimes purely by the compiler) there's the possibility of being able to interleave somewhat dependent work as additional code within a Kick rather than as a separate subsequent Kick, as in the diagram below:

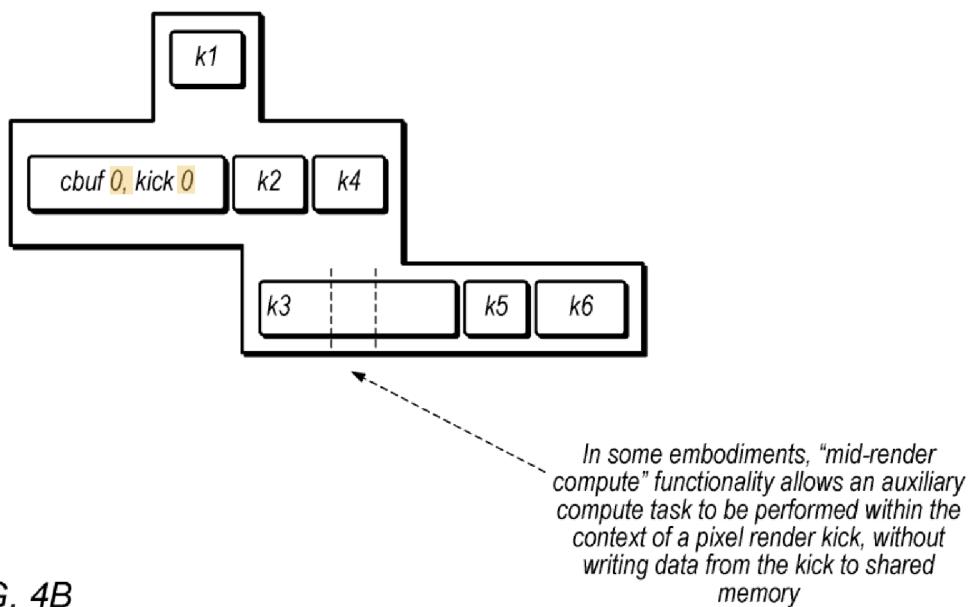


FIG. 4B

Two things are going on above. The basic GPU model (somewhat hardware-related, somewhat software-forced) includes two parts

- automatic synchronization occurs at kick boundaries AND
- kicks are of a particular type, associated specifically with either the special purpose vertex hardware,

or the special purpose fragment hardware (or likewise ray hardware) or purely compute.

This sounds good, but consider what it implies. Fragment processing (ie convert each triangle *within a tile* into scan lines of pixels, and shade [ie texture] each pixel) has to be completed for the entire kick, meaning the entire image, before a subsequent compute pass that wants to modify that image in some way. (In the simplest example, think of something like I want to generate a 3D image, then apply a blur to it.) This is obviously sub-optimal in terms of memory and performance.

So we bend the model slightly! We introduce a new type of shader called a *Tile* shader which is allowed to execute “within” the fragment shader kick as a subroutine, even though in some sense it “should” be part of a different, Compute Data Master, kick. The Tile shader can execute after the fragment shader is done shading this particular tile, reusing data in the Local Image Buffer, and without the Tile being written back to any sort of shared memory.

If this is possible, then why bother with these three “tracks”, one for Vertex Processing code, one for Fragment/Pixel processing code, and one for Compute code? The three (as I keep repeating, four in the case of M3) tracks correspond to conventions (three queues) in how the CPU+OS communicates with the GPU’s firmware. But why bother if, as we see in this patent, the system seems OK with having what would be considered “Compute” functionality slotted into the Pixel track?

There are at least two reasons:

- the system tries to schedule simultaneously kicks from all three tracks, since, even if all three are, to some extent sharing the Shader Core and Load/Store, it’s obviously better in terms of performance to have Vertex, Fragment, and Shader Core all working simultaneously.
- there are opportunities for energy reduction, eg if we know that all the currently active kicks on a core are Shader only, we can shut down Vertex, Fragment, and Ray until an appropriate kick begins.

the local image buffer (fancy Tile memory)

We also get a nice picture of what the Local Image Buffer looks like. Tile sized, with the obvious RGBA channels, but also at finer resolution than the ultimate screen image, to allow for MSAA.

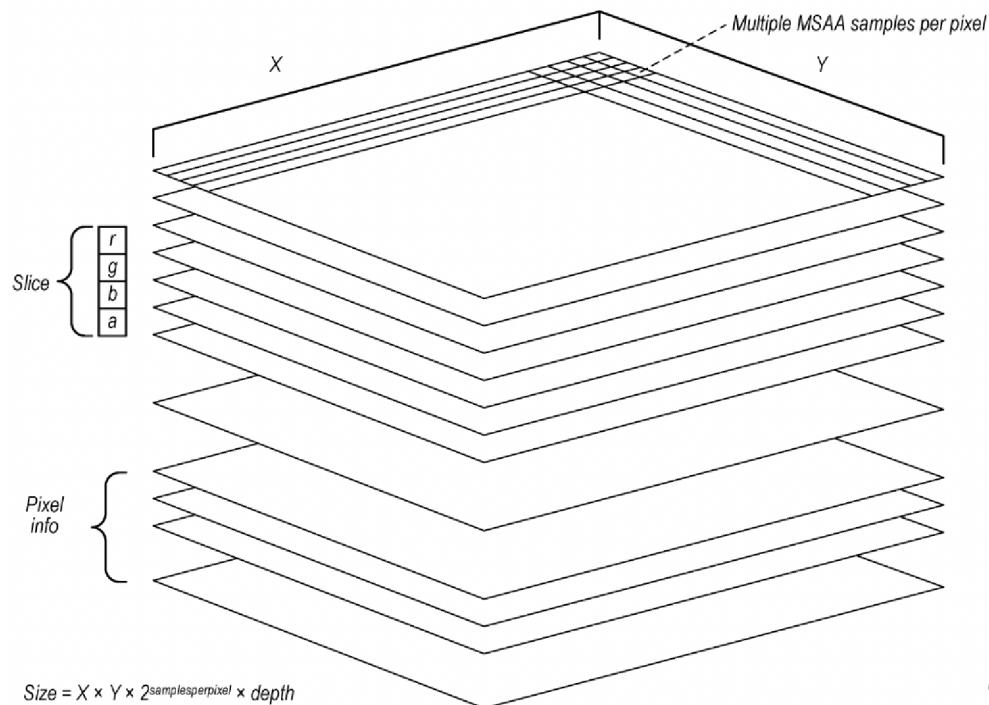


FIG. 6

Another way to say all this, being a little more explicit about the parts that work because of specific Apple choices:

How to achieve the desired goal of executing compute/render code interleaved with pixel code? Suppose we are able to enforce the following rules, something like

- all pixel processing associated with a given *tile* of the screen occurs on one fixed GPU core (different tiles for different cores)
- the compute/render task of interest is likewise scoped to that same tile
- we provide a dedicated address space plus storage for that tile (the so-called *Local Image Buffer* or LIB).

This won't match all mid-render passes, but will match many of them.

Now, if we ensure that pixel processing and the render pass both read/write their shared data to the Local Image Buffer, then the problem of synchronization becomes a lot easier. The data shared across code [fragment shader then mid-render-compute] only needs to stay within a single core (since it's all scoped to a single tile, no need to flush to L2, or even out of Tile storage); and since it is within a *specific address space* (Tile storage/Local Image Buffer) which is not shared with other GPU cores or the

rest of the SoC, and so defined to be *ephemeral*, there is no necessity to worry about flushing it to L2 after the Kick is complete. The only thing we (actually compiler) needs to ensure is that the RGB(and possibly A) planes of the Local Image Buffer are flushed to global storage at the end of the Kick.

All this means the compiler can do even more of the job behind the scenes. As long as we clarify what data is ephemeral (and so should be placed in the Local Image Buffer) the compiler can make sure that the appropriate load/store commands for that address space are used.

(As we've already said, the LIB is configured as four dimensional storage (X by Y pixels, each pixel has C channels [like RGB], and each channel has B bytes of data associated with it), then the hardware more or less automatically handles the address details.)

There is however also an escape hatch just in case you want to take advantage of this LIB functionality but also want the final result (a texture or whatever) to be used by subsequent GPU processing; in that you can DMA the appropriate (possibly additional) channels of the LIB out to the L2 shared address space, rather than just relying on the default graphics path writing out RGB.

ordering control for the local image buffer

Finally a third optimization is described which takes advantage of the LIB.

Think of the problem of drawing graphics. Suppose all we have is a list of triangles to be drawn. And suppose we simply start drawing triangles (ie decomposing them into fragments, and shading each pixel of each fragment) with no additional planning or co-ordination.

This will result in some front triangles being overdrawn by back triangles!

We need some mechanism to sort these triangles by Z-order to ensure that we never overdraw triangles that are foremost.

There have been many algorithms to do this, but graphics hardware has basically converged on the Z-buffer as the optimal solution for this. As a quick reminder, this means that along with the already mentioned RGBA planes of a tile's Local Image Buffer, there will now be an additional plane that holds the Z-value (distance from the viewer) of the pixel drawn at that spot. We can now handle every pixel independently (which is what we want! maximum exploitation of all the lanes of a shader, all running interleaved warps) subject to the rule that

- when I write the final output of a pixel, I write not just RGBA but also Z
- the hardware write circuit compares this Z to the previous Z stored in that pixel
- if the new Z is closer to the viewer, the pixel is written. If the new Z is *further* from the viewer (so *behind* the already stored pixel value), the new pixel value corresponds to a behind triangle and is discarded.

This seems like a great plan and does actually work very well. But it can't handle everything. Specifically, it can't handle situations where a front triangle is partially transparent, and so what should be seen depends both on the value of the front triangle and having the value of the back triangle already present and drawn (so we can combine the back triangle value with the front triangle value).

The general solution to this is called Pass Groups. Triangles are bundled (perhaps by the developer,

perhaps by sorting hardware before fragment processing) into a sequence of Pass Groups such that Pass Group 1 is drawn first, then Pass Group 2, and so on.

As far as correctness goes, this works, but it introduces a forced sequential processing into what we'd prefer to be a fully parallelized operation; and as we've seen repeatedly as we've discussed scheduling at every level from kernel to warps, we'd always prefer to replace forced sequential code with parallel code. So can we do this here?

Yes! With something that looks somewhat like a Z-buffer.

The patent describes this in very general terms. My explanation below is much more specific but, I think gets at the essence, more or less, of how it's implemented, and thus what are the strengths and limitations. Of course I may be completely wrong in every aspect of this explanation... (In particular the way the patent talks about `pixwait` doesn't make sense to me, compare to the scheme I describe...)

For simplicity imagine the problem as three pass groups, each drawing one triangle, so that triangle 1 needs to be drawn first, triangle 2 partially transparent, partially on top of 1, and triangle 3 partially transparent, partially on top of 2. Also for simplicity assume just two threads of execution.

We could just draw triangle 1, then triangle 2; no parallelism, only use of the first thread.

But suppose that we have a small amount of state, just a few bits, call it say a P-buffer, associated with each pixel. Now we use the following algorithm

- 1) Run the first pass group simply marking in the P-buffer every pixel that will be drawn by the first pass group. This should be fast because there is no shading, just generating which pixels are touched.
- 2) Run both pass groups simultaneously. When we write to a pixel, we first read the P-buffer to see the pass that "owns" the pixel. If no-one owns it, or if it's owned by the first pass and the first pass is writing, then the write goes ahead. If the pixel is owned by the first pass but written by the second pass, then the P-buffer marks an additional state bit of "needs to be fixed up by pass 2", but nothing gets written.
- 3) At the end of this (where, mostly, pass 1 and pass 2 could independently write all their pixels) there are a few pixels marked as "needs to be fixed up by pass 2". These are now "owned by pass 2" and we more or less repeat what we did before.

Pass 2 now runs again in parallel with Pass 3. In the simplest version Pass 2 runs fully and overwrites pixels it already touched; in a fancier version we have auxiliary storage that can be used so that Pass 2 only calculates values for, and writes to the "needs to be fixed up by pass 2" pixels. Meanwhile Pass 3 is doing its thing, and is now locked out of writing to "owned by Pass 2" pixels but can write anywhere else (including transparently blending with all the pixels that were successfully set by Pass 2 at step 2). Pass 3 can't, however, yet write to the "owned by Pass 2" pixels, so those get another bit set as "needs to be fixed up by pass 3".

- 4) And so it goes. This can be repeated indefinitely, with just a few bits in the P-buffer for each pixel, since at any given stage, only two or three Pass Groups are of interest, and bits can be reused as we pass from one set of Pass Groups to the next.

At a more abstract level we can consider the P-buffer as providing something like a very high speed per-pixel lock. Not very flexible, but good enough to perform the basic tasks of

- lock the pixel to one writer task, and prevent others from writing it
 - note that other task(s) did want to write the pixel (to allow for a subsequent stage of fixup)
- With this primitive you can do various things that involve running in parallel two (or possibly more?) otherwise sequential blocks of code with, hopefully, a very small amount of final stage cleanup of work that could not be parallelized.
- Blending is the obvious use case, but another example might be something like constructing fake shadows.

It's an interesting question whether this could be used for other things...

The problem has to have the same form (a sequence of passes, which would run much faster if run simultaneously but for occasional, but rare, write collisions); but if it does have that structure, you may be able to map all sorts of interesting use cases onto what is effectively a large array of very fast simultaneous locks.

(2019) transient cache

(2019) <https://patents.google.com/patent/US11023162B2> *Cache memory with transient storage for cache lines* describes a very strange setup.

The idea seems to be that within some cache (probably the GPU L1) each cache line has some additional associated storage which we can think of as associated with the address of the cache line, but living effectively in a separate address space, so that when the line is removed from cache this additional storage becomes invalid (is not written out anywhere) and when a line is loaded into the cache the extra storage remains invalid until deliberately set.

The generic point seems to be that we occasionally have situations where, for *some* purposes

- data structure A is used together with separate data structure B
- we have good reasons not to merge the two into a single data structure
- but at other times we want to use the two together.

So we, effectively, load data structure A (large) into the cache and the (small?) part of data structure B of interest in this extra per-line storage, and we have the two available for our purposes.

This all seems predicated on using the L1 "cache" much more as a controlled local storage than as a blind cache (where data could at any time be removed), and the various examples described in the patent (which don't make much sense to me) seem to match this idea, specialized use cases by the GPU firmware.

One such case has to do with processing lists of vertices; another has to do with cases of profiling or debugging where the additional storage can hold useful data associated with the "primary" data.

Overall I think this is much more a very low-level "here's a way we solved a problem in this particular chip" type of patent, than any sort of indicator of the future, especially the future as relevant to developers and outsiders. It probably makes sense for Apple to provide some extra (transient) storage for debugging and performance monitoring along with some instructions to read/write that storage. Then

it makes sense to link that storage to L1 SRAM. Then, maybe, it make sense for GPU firmware writers to see that it's available and think "hah, if we used that debug/performance storage to hold these few temporary bits, we could make our linked list traversal faster [or take up less cache space]". But this final step is just a clever hack to exploit HW that's already available for a very different purpose?

some numbers for gpu caches and TLB

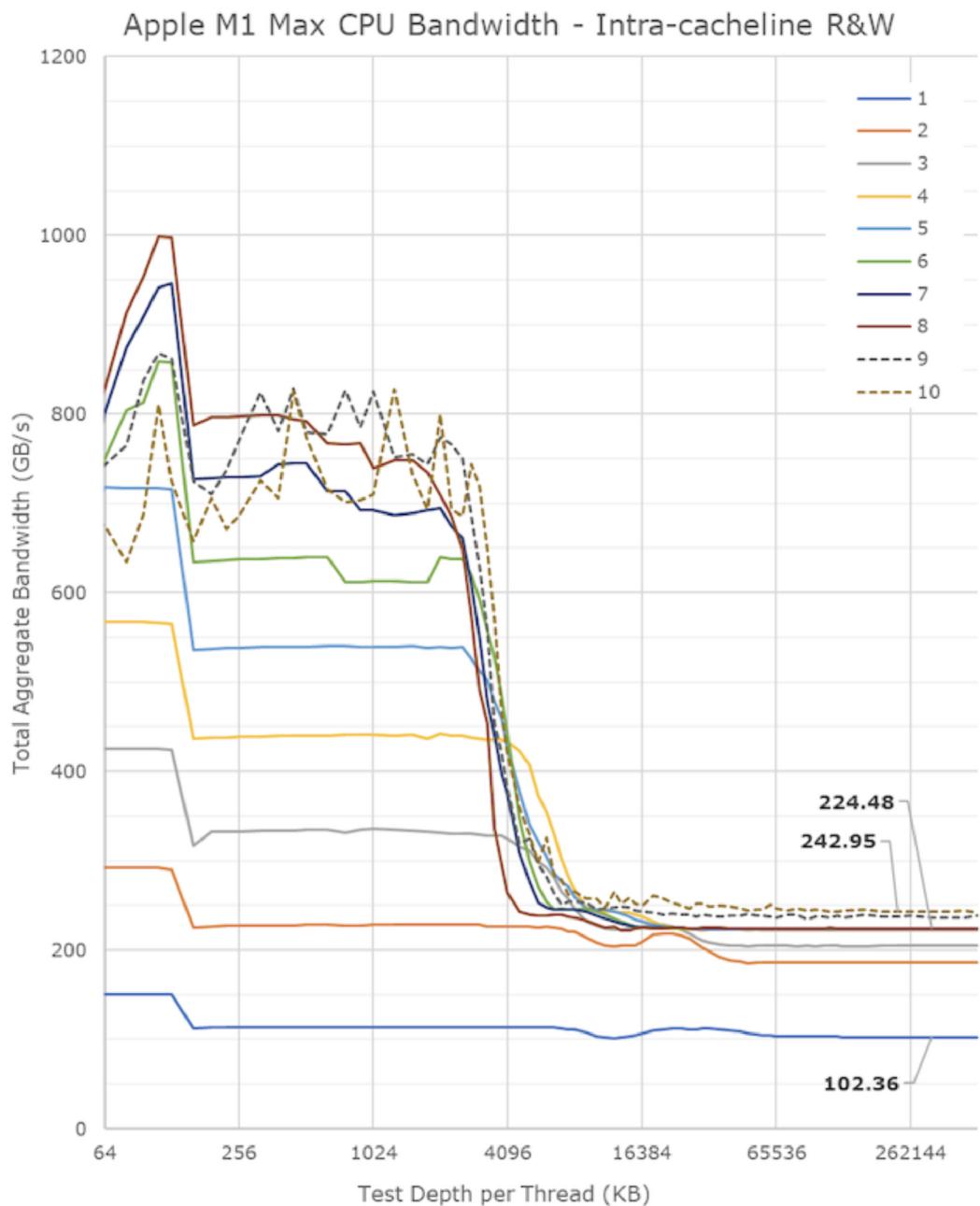
reminder about the CPU

At the most basic level, the GPU has a per core data L1 and global L2 that are more or less like their CPU equivalents. But "more or less" does not mean there's no scope for optimization. Consider, in particular, what the cache line length should be.

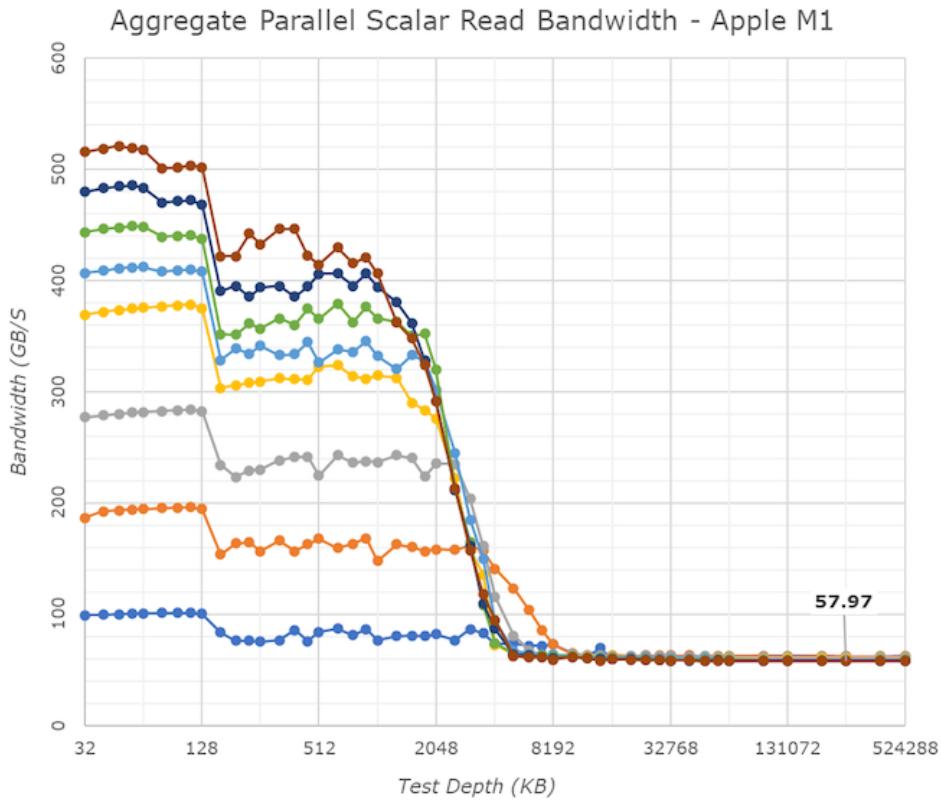
Apple seems to have (as far as I can tell) settled on 128B as the system-wide line length, with the CPU optimizing for a shorter line length of 64B. So we get a CPU L1 line length of 64B, and a CPU L1 line length of 128B, the two connected by a 32B wide point-to-point connection, and with an "effectively" 32B wide cluster-to-SLC connection (possibly implemented as 64B wide running at half the CPU's ~3.x GHz).

This gives us the throughput results we have seen before, from Anandtech <https://www.anandtech.com/show/17024/apple-m1-max-performance-review/2>.

(Essentially for a single thread the L1 max number is three 16B loads per cycle, the L2 number is a 32B load per cycle, extending all the way out to SLC. We see the L2 number manages to keep climbing for multiple L1 to L2 transactions [separate point to point connections]; while the the L2 to SLC number climbs for two clusters (with a small extra boost for E-cluster) then maxes out.)



We'll also want to (for reasons that become clear) see the M1 (not Pro or Max) CPU bandwidth, from <https://www.anandtech.com/show/16252/mac-mini-apple-m1-tested>



numbers for the GPU

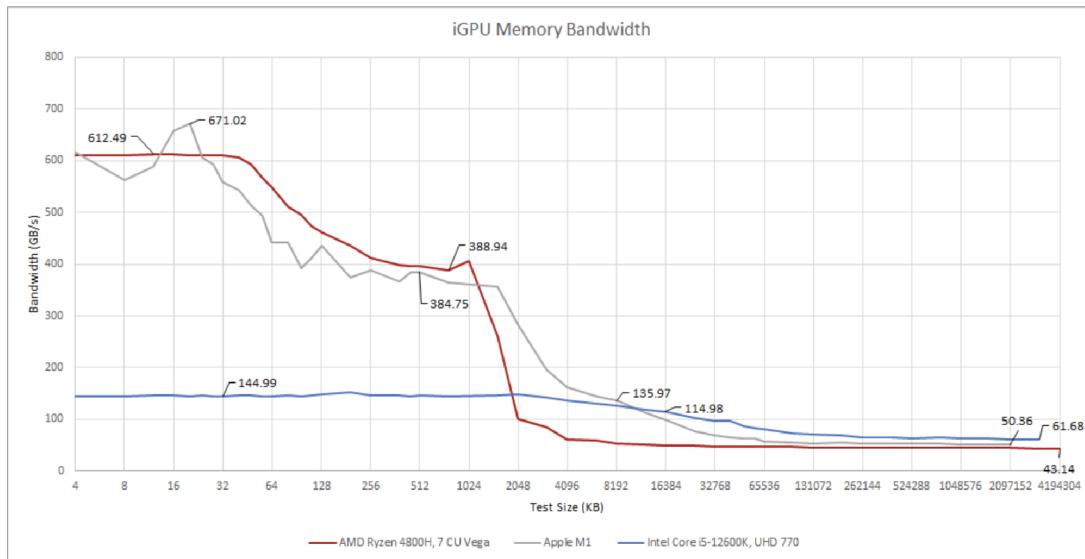
In terms of cache lines or page sizes, the GPU ideally probably wants to go in the other direction, in that most GPU data is part of a fairly large data structure, most of which will eventually be accessed. So you can save a few transistors and some energy overhead by using longer lines, so that a single cache tag covers a lot more data.

On the other hand, there may occasionally be short lines, and the GPU L1 is very small, not much space to waste in unused line length.

My guess (just a guess!) is that the GPU line length is something like 64B for L1, and 256B for L2.

Latency tracking, see <https://chipsandcheese.com/2022/05/21/igpu-cache-setups-compared-including-m1/> suggests L1D is 8KB per core, while L2 is 1MB (shared across 7 or 8 cores for an M1).

The comparable graph for bandwidth looks like:



Does this make sense?

The DRAM bandwidth is about 50..60 GB/s, same as the CPU bandwidth (on an M1, not a Pro or Max), so maxed out by the DRAM's intrinsic performance. Within the SLC range (so from about 1 to 9 MB) even at worst case we're twice the DRAM bandwidth. This suggests a 64B wide "effective" data connection from SLC to GPU, again perhaps implemented as 128B wide running at half CPU frequency.

The GPU is claimed by the internet to be running at $\sim \frac{1}{3}$ CPU speed, but the L2 bandwidth appears to be $\sim 3x$ the SLC \leftrightarrow L2 bandwidth, so overall something like $8 \times 64B$ paths out of L2 (ie per core 64B path from core to L2)?

The L1 then suggests something like 128B ($32 \times 4B$) paths from L1 to the shader cores, so effectively each quadrant has the opportunity to read a full dataset (32 lanes of 4B registers) every fourth cycle.

L2 (and TLB) multiblock tagging/fetching

The question now is, given all the above, how best to implement the L2 and its communication with SLC. We want wide L2 lines (multiple of 64B), and we want efficient communication with SLC.

One option is on a miss to a cache line, to load in the full 256B. This works and is probably not a bad solution, but will occasionally be suboptimal, paying the energy cost of moving around unneeded data.

A second option is to use a sectored cache (something we've already described) so that the single 256B tag sits along side four valid bits that each cover 64B of the full cache line. Now each 64B line is loaded on demand, otherwise left invalid. This is also not a bad solution, but we can do better.

What (2021) <https://patents.google.com/patent/US20220374359A1> *Multi-block Cache Fetch Techniques* suggests is that we start with a model like the one where a single 64B sector of the line is requested, but as that request sits around in the L2, waiting for its chance to be placed on the NoC and sent to the SLC, we track further requests that come in that cover other sectors of the same 256B line, and if appropriate, add them to this request. Eventually the request that is sent out to the SLC is a single NoC transaction that requests anywhere between one and four cache lines depending on the activity level for this particular line. This is a nice, but simple, way to achieve the best aspects of both the two obvious options.

The same idea could obviously work for the CPU L2 cache but perhaps is not as valuable because a GPU generates so much data traffic that there will often be a fairly deep queue of requests out to SLC, allowing time for multiple misses to the same (long, eg 256B) L2 line to aggregate; for a CPU these sorts of deep queues should be much more rare. On the other hand, CPU prefetching may be confident enough about the future to make it worth generating a single longer transaction from CPU L2 to SLC?

Now consider TLBs.

<https://github.com/AsahiLinux/docs/wiki/HW%3AAGX> claims that on M1 the GPU TLB looks like

- L0: 2 entries
- L1: 8 entries
- L2: 2048 entries
- L3: 2048 entries

(It's unclear how these numbers were measured. In particular it's very unclear whether they represent total coverage vs number of distinct coverage points. This becomes very important because it's highly likely that each entry corresponds to multiple possible successive pages.)

Possibly the L0 is even per quadrant, if there are four independent load/store units (something still unclear), and/or with separate L0 storage for loads and stores, but a common L1.

(As a technical matter, the Companion Core uses a UAT Unified Address Translator. This is in, in some technical sense, different from the CPU TLB, or from some other TLBs used by other companion cores, which use IOMMUs or SARTaddress filters. I'm not very clear on the exact differences between these variants.)

So what do we mean by multiple successive pages?

Taking the previous L2 cache line aggregation idea further, the TLB of a GPU likewise probably has an

ideal “fragment” size that is larger than a CPU’s single page, so we can do the same sort of thing as we did with the cache line.

Imagine, for example, the tag in a TLB covers not just a single page table translation but four successive such translations, indicated by four validity bits. Now, without much effort or much additional energy, we’ve extended the reach of our TLB by four, in a way that probably works very well for GPU

If we do this (and the patent implies that we do) then on the occasion of a TLB miss, we can do the same sort of thing – while we wait for the TLB miss to be serviced by higher level caches and TLBs, we can check to see if subsequent TLB misses are served by that same TLB tag (ie are in the group of four TLB entries all serviced by the same tag), and once again a single outgoing request to the L2 TLB can ask for anything from 1 to 4 TLB mappings to place into the L1 TLB.

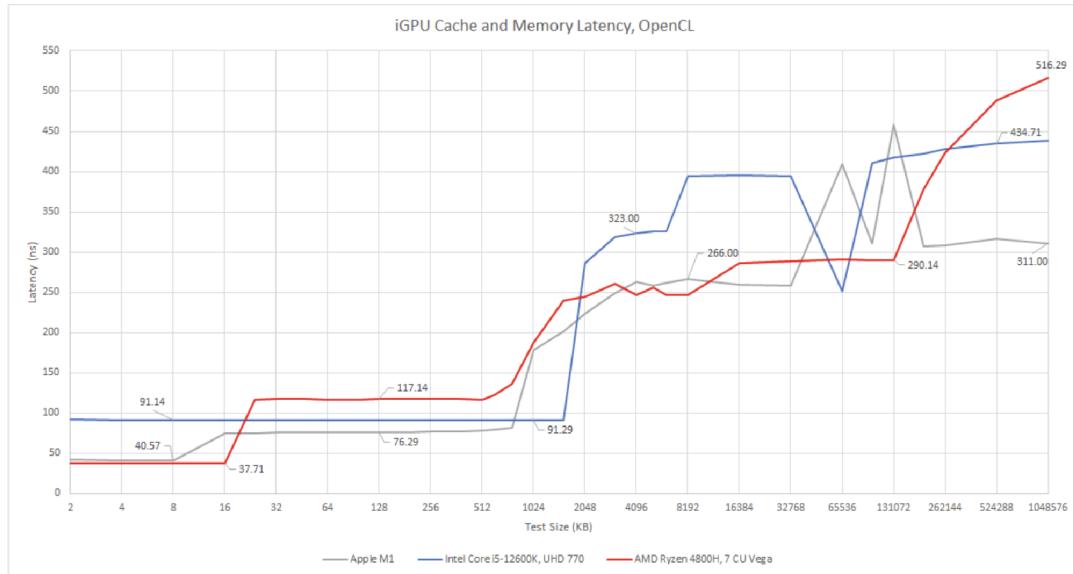
(There are technical details here that limit this from being doable for certain situations, eg if successive pages change their permissions, or if successive virtual pages map to non-successive physical pages, but the OS can lay things out to minimize these issues.)

(Note also that in both the cache line and TLB cases, what we are doing is *not* prefetches; these are requests for data that is definitely needed; but they exploit specifics of how GPUs work, from the large data streams to the way that so many requests are constantly being queued up and held temporarily until each gets its chance at access to the NoC.)

We can look at this from another angle.

2048 entries each of 16kB covers 32MB, and the ChipsAndCheese latency graph below shows a (noisy) jump at 32MB, with lots of subsequent variation. An obvious first guess is that the L0, L1, L2 TLBs are within the GPU (L0 and L1 per core, L2 at the GPU L2 level), and the L3 TLB is at the SLC level. The behavior from 256MB to 1GB shows that something clever manages to hold the latency down to not much more than the guaranteed L2 coverage [2048 entries], so is that coalesced pages in the L2 [32 pages per entry] or coalesced pages in the L3/SLC TLB [again 32 pages per entry]?

(Like the case of Asahi, we have no idea exactly how ChipsAndCheese are performing their tests, so while this *probably* means that the L2 TLB can cover at least 32MB, it doesn’t settle the issue of whether coalesced pages can cover more than 32MB, and you could argue that the non-monotonic behavior shows we’re occasionally exploiting coalesced pages, depending on exactly how the benchmark operates.)



L2 arbitration (cheap way to fake oldest-first arbitration)

At some point memory requests have to be sent to L2 and then on to SLC/DRAM. Consider specifically L2, which is, I think, the context for this patent (2021) <https://patents.google.com/patent/US11443479B1> *Snapshot arbitration techniques for memory requests.*

Suppose we have multiple (and possibly a large number) of queues, eg per GPU core, each submitting requests to L2. I'm thinking of, for example, an M2 Max with each GPU core (up to 39 of them) having a queue of requests for the L2. Each cycle only one (?) or at least some small number of requests can be sent on to the L2.

This is a classic scheduling/QoS problem: each cycle which queue gets access? (We assume that once a queue is given access, it submits the first request in the queue.)

Easy and low-power solutions that work “well enough” when the system is not too overloaded include

either round robin from each queue to the next, or simply random choice amongst the non-empty queues. We could handle priority (if that were important) with something like two queues per core, and weighted round robin, so that eg every high priority queue gets three requests before we switch to a low priority queue.

But these sorts of ideas perform sub-optimally as the machine gets more loaded. A possibly implementable algorithm closer to what we'd like might be "oldest first" across all the queues, but if you think about that, that involves storing a timestamp for each entry in each queue and moving those timestamps fast enough to a central arbiter location, and constantly comparing them. It might be doable, but not at low power.

However, can we *approximate* oldest first? That's what the patent does.

Simplifying (as always!) imagine something like every 20 cycles or so we take a snapshot of the queue states, meaning we assemble a vector that records how many entries are sitting in each queue. When we compare the oldest snapshot with the second oldest, we get a count of how many values were added into each queue during the 20 cycles.

Assume we now jump into the middle of the algorithm, at a point where we have just finished handling (ie passing through the arbitrator) all queue entries associated with the oldest snapshot. That means we effectively have a list of how many entries, in each queue, are associated with the 2nd oldest snapshot and are thus (approximately) the oldest entries.

We can then now round robin across all the queues that are non-empty with respect to this interval, and each time a queue gets a slot we subtract one value from an associated queue count. Proceeding in this way until all counts are zero, we won't provide perfect oldest first, but we *will* ensure that all entries submitted during this oldest interval of 20 cycles are handled first, before we move on to the set of entries that arrived during the next 20 cycles. We can readily adopt this idea to other specific situations, for example always handling a higher priority queue first, but using this scheme to handle lower priority requests in more or less oldest-first order.

TLB Issues

background

When it comes to the "traditional" TLB, there's not much to say. We've already discussed the main points

- a graphics TLB needs very large coverage, so nVidia uses a combination of 2M and 32MB pages. Apple probably uses coalesced pages to get at least somewhat larger coverage, along with an MMU that caches multiple layers of the page table (especially a large number of the penultimate level entries) and storing page table entries in the SLC as a special class, so that page table lookups are reasonably fast.
- nVidia, for whatever reason, use a virtual L1 but physical L2, and so need a TLB sitting between L1 and L2 caches. Apple only translates to physical when leaving the GPU, so L1 and L2 are both virtually addressed, and substantially less TLB bandwidth is required.

the “graphics” TLB

But that's not the end of the story because Apple uses a third layer of address space.

We've covered this before, but the idea is sufficiently novel that we'll cover it again, now from a slightly different angle.

To get started, let's remind ourselves how things work on a CPU. In particular, imagine a virtually addressed L1D cache, a so-called VIVT cache. (Virtually addressed CPU's are not common, for various technical reasons, but they are not impossible, so let's assume such).

Some points on interest in such a design:

- Virtual address space is effectively unlimited. The OS can hand out to a process vast amounts of *virtual* address space, page after page, and as long as the process never even accesses that virtual address space there is no problem. Thus we can do things like give the process a huge effective stack, a huge effective heap, lots of space for shared libraries, etc. All of these large address space allocations are very convenient (use deep recursion, for example) but we never have to pay a cost for them until we actually use [ie access] some of the address space. This is much more convenient than earlier address allocation models (like say macOS prior to System 7) where an app was given a fixed allocation of *physical* address space (ie a fixed allocation of RAM) and was either in trouble if it turned out to need more, or kept that address space from another app that might be able to make use of it.

- The actual physical storage referenced by this virtual address space works because we use multiple levels of caching. In particular the L1 cache doesn't hold any specific set of addresses, instead each line of L1 cache holds some data (referenced by matching an *ASID+virtual address* to a cache line tag) that is currently of optimal use. Later that data may be less useful and moved to L2, or out to DRAM, or even out to disk.

The virtual address abstraction allows us allocate lots of *potential* (but generally unused) resources; the cache abstraction (ie address L1 or L2 lines by matching line tags, not by hardwired addresses) means that we can get to the data of current interest very fast, and we can continually and transparently swap the data of current interest vs no-longer of interest between different cache levels.

- The fact that we are using a virtual cache (VIVT) means that we usually don't even have to pay translation costs. We have to access a TLB when transiting from the L1 cache to the L2 cache, but those accesses are far less frequent than accesses within the L1.

- Processes can operate with as much virtual address space as is useful, and can convert that into as much physical address space as is available. They will pay a price as they start using more address space than fits in first L1D, then L2, then SLC, then DRAM, then eventually flash/disk. But the transition from level to level will be slow, and things will continue to work. Even so, the OS will track what is going on and, if there is too much activity swapping to flash/disk, will either suspend some processes (so that fewer are executing simultaneously using up all the RAM) or even kill some processes.

Now let's think of the same problem within the context of the GPU.

The traditional GPU model (ie the Apple model up to and including the M2) is a few fixed sized fixed address spaces, for example register address space as one part of per-core storage, and Scratch-

pad address space as another part of hardwired per-core storage. This is analogous to the Mac System 6 memory model where an app/kernel is given fixed allocations for various tasks (fixed sized stack, fixed sized heap, etc), and so has to ask for the maximum it might possibly require so that it doesn't run out, and then is frequently “wasting” those resources [withholding them from clients that could better use them].

Suppose we apply the previous ideas to this problem. To simplify the discussion, for now let's limit the example to allocating Scratchpad storage.

Instead of allocating a hardwired range of Scratchpad SRAM cells (ie a range of physical addresses) we engage in two levels of abstraction. The first level of abstraction is we give a kernel a range of *graphics* address space. This is analogous to a CPU getting virtual address space. So the kernel gets say 64kB of graphics address space, which is simply some entries in a lookup table, it has no effect on physical resources. Heck, if we want to give the kernel instead 640kB of graphics address space we can do so; it's just a few more entries in some remapping table.

What next? The most obvious thing you might think of is we map (via some special lookup table) a graphics address into some range of per-core SRAM, ie we translate a graphics address into a physical address. But that is sub-optimal, it is ignoring the second of my points above, the point about caching. Instead we use cache-based ideas. Each line in the SRAM is tagged with a *kernelID+graphics address* and as we access addresses in our allocated graphics address space, we either hit existing allocated lines in the per-core SRAM, or we allocate new lines.

What happens when we run out of per-core SRAM lines?

Well in the CPU case, we started moving lines between the (virtually addressed) L1D and the (physically addressed) L2, and, if necessary, out to DRAM.

In the GPU case, we start moving lines between the (graphics addressed) L1D [ie per-core SRAM] and the (virtually addressed) L2, and if necessary out to (physically addressed) SLC and DRAM. Suppose that we have, say, 64kB of SRAM per core. We can still allocate the kernel 640kB of graphics address space, and if the kernel only uses 64kB or less, every access to Scratchpad (via a graphics address) will in fact hit in the per-core SRAM. And the access will be via a graphics address and tag lookup, not via a TLB, so we also won't pay any TLB latency lookup costs. We'll be no worse off than before. But we have flexibility! If we actually want to use 96kB of the allocated graphics address space, then at any given time 32kB of our Scratchpad lines will reside in GPU L2, but hopefully the caching scheme works, and most of the time the active lines are actually the lines present in the local SRAM. The lines residing in GPU L2 will be addressed by virtual address, having been translated from graphics address by an rTLB sitting between the GPU core and the GPU L2. In principle there is no reason why those Scratchpad lines couldn't even, after a long period of not being used, be discarded from L2, to be translated from virtual address to physical address (TLB sitting between GPU and the rest of the SOC) and placed in SLC. Then later accessed and pulled all the way back from SLC to per-core SRAM.

The same idea holds for registers. We can allocate each warp the full set of 128×32b registers (which is equivalent to 16kB of storage, recall each register is 32 lanes wide) just by handing out free graphics address space pages. Each warp can then translate any register lookup from (warplD+register number) into (*kernelID+graphics address*) to access the registers in the communal pile of per-core SRAM; basi-

cally like how any CPU with a standard (ie physically addressed) L1D passes each load/store through the TLB before accessing the cache .

This might seem expensive, but we have two things that help us.

First is that most register accesses now take place through the operand cache. So, just like the TLB sitting between GPU L2 and SLC, just like the rTLB sitting between GPU core and GPU L2, we are able to perform most accesses most of the time without requiring a translation.

Second is that our instruction pipeline mini-front-end scheme can handle most cases of moving data between the per-core SRAM and the operand cache simultaneously with other instructions that are currently executing and getting their operands from the operand cache. So an instruction is rarely delayed by the extra cycle or few of translation of (warplD+register number) into an address, and then the access of that address in the per-core SRAM (acting as a large cache).

There's one final technical point. We want the single per-core pool of SRAM also to act as a traditional L1D cache. So we want this pool of SRAM to hold ~384kB of registers (ie the amount of space M1 devotes to registers), and ~64kB of Scratchpad, and ~8kB of L1D, and some specialized storage for vertex processing, fragment processing, polygon sorting, ray tracing etc. Probably about 512kB in total? So this pool of SRAM is acting as a cache for all these different use cases. Most of these items are found in the cache by the usual set/way/match-tag business. Most of these items are living in graphics address space – but not all; in particular lines that are being used as L1D cache are addressed by (ASID/contextID+virtual address), not by (kernelID+graphics address). But this is no real issue; a tag match is just a bit match regardless of how we interpret the bits. We add one more bit to the tag to indicate that these bits correspond to (contextID+virtual address) rather than to (kernelID+graphics address) and everything else just works.

Also almost all the previous resource allocation complications go away. Most of the resource allocation complications were a result of having different types of resources, so how do we trade things off if kernelA wants lots of Scratchpad whereas kernelB has warps that require lots of registers? Now all of these (Scratchpad space, register number, etc) all get translated into a single metric of “lines of SRAM actually allocated” and these dynamically and transparently move between SRAM and L2 depending on how important they are. The main thing resource allocation now needs to do, like an OS, is track that SRAM lines do not get too oversubscribed, and pause warps or threadblocks when it looks like too much swapping between SRAM and L2 is occurring.

I've described all this by analogy with a CPU and an OS, and this seems to be much how Apple thinks of it.

The patent (2020) <https://patents.google.com/patent/US20230385201A1> *Private Memory Management using Utility Thread* essentially describes the “OS” aspect of it and is pretty obvious. The MSL compiler attaches to a kernel a description of the maximal resources (Scratchpad, registers, etc) the kernel may require. Then a dedicated thread running on the companion core, acting like the Virtual Memory subsystem of an OS, takes that list of resource requests and fills in a “page table”, ie a mapping between the graphics address space given to this kernel, and the virtual address space of the process

that submitted this kernel. With those page tables in place the hardware just runs as I have described, then at the end the kernel releases the resources, ie the mapping table and the range of virtual address space that was mapped against this particular graphics address space.

The patent (2020) <https://patents.google.com/patent/US11521343B2> *Graphics memory space for shader core* describes more of the “hardware”, less of the “OS” side of the operation.

using the TLB to enforce hazard ordering

Now consider a different issue.

Whenever you have out of order execution you have the potential for hazards, ie reads that happen before writes, or writes to the same address in the wrong order. In a CPU we handle this

- for registers via renaming
- for memory addresses via the load/store queue

What about for a GPU?

The first point is that GPUs make no promises about accesses to the same address in terms of ordering of threadblocks, or in terms of access by lanes (eg if two lanes in the same warp write to the same address). The most common way to handle cases that need to be handled is by a fence, which is usually implemented by ensuring that all warps hit a particular instruction, then starting execution past that instruction. Meaning the job is essentially handled within the core.

Beyond that, GPUs are in-order! So why is this even a concern?

Well are there places where execution has slip out of order? Within a datapath clause, instructions cannot begin until operands have been collected, and a subsequent instruction cannot begin until the previous operation has completed. So I don't see any obvious problem there.

Even within a load or a store clause, instructions from a single warp happen in order, eg instructions are queued in order to arbitrate for cache access. So there doesn't seem to be a problem there. Well...

Suppose a successive store then a load (back to back clauses) or two stores (from the same clause) access the same address from the same lane; for example this could be a Scratchpad address. If the access hits in local SRAM I don't think there is a problem. So suppose access misses local SRAM. That means the address has to be translated by a TLB-like structure from graphics address to a virtual address, to access the L2. This still is not a problem as long as requests to the TLB are handled in-order (at least within the same warp) which is easy-ish as long as the lookup hits in the TLB. But what if the lookup misses in the TLB? Then conceptually what we want to do is place the request to the side so that other requests can be serviced by the TLB until the translation lookup (by accessing the equivalent of the page table for the graphics to virtual address space translation) is performed.

The simplest way of implementing this TLB access builds a list of these waiting requests in a way that enforces the correct ordering; but it enforces this ordering across all requests, not just requests from one warp.

A slightly fancier version uses multiple such lists (eg 8 lists, based on a hash of the graphics page address being looked up) which gives some degree of parallelism (we still have to wait for all the

requests ahead of us before we get processed, but we only have to wait for the requests in our queue to be processed, we can ignore the 7/8th requests in the other seven queues).

But Apple thinks this still isn't good enough, hence (2021) <https://patents.google.com/patent/US11467959B1> *Cache arbitration for address translation requests*. This is one of those patents where I'm really unclear quite what they are doing; this is all my best guess!

Their primary concern appears to be, as usual, QoS. They want a scheme that still enforces warp order for load/stores within the same warp, but allows arbitrary reordering otherwise, in particular so that high priority requests waiting on the TLB (and specifically waiting for their TLB miss to be serviced) can be handled immediately once the miss returns.

The trick is the structure below, in particular the Hazard CAM. The Hazard CAM records enough about the previously queued (and still waiting) requests that a new request can come in and see if it matches an earlier request in the Hazard CAM (is trying to access the same address at some granularity; perhaps page granularity, perhaps cacheline granularity.)

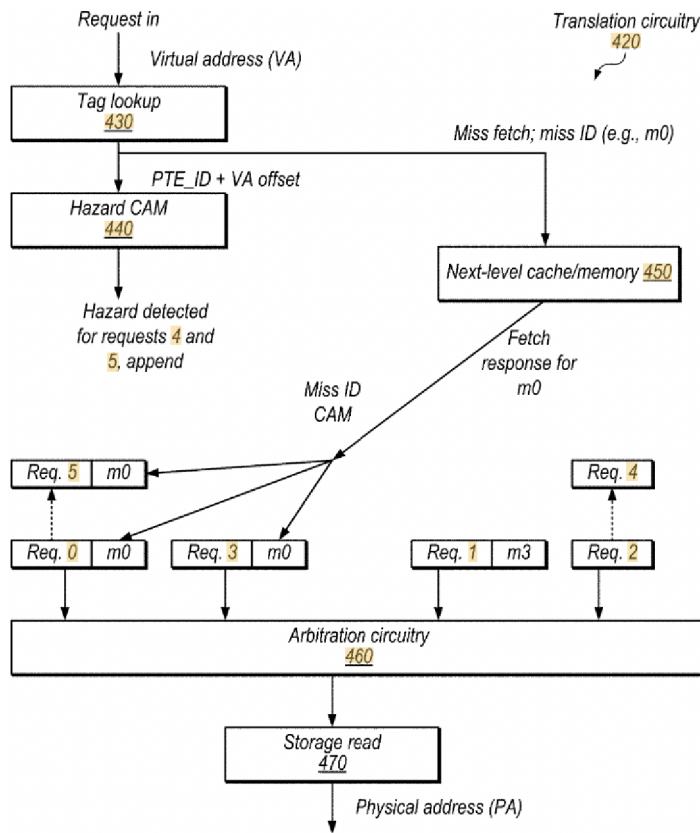
If there is a match, then the new request is placed behind the matching earlier earlier request; otherwise the new request becomes the head of a new queue. We now have some large number of queues (eg those with heads Req 0, Req 3, Req 1 and Req 2) and we can arbitrate based on the priority of each head, with no more enforced ordering than is demanded by the actual code.

The example as given shows that each of Reqs 0, 3 and 5 are to be serviced by the same miss, miss m0, but if Req 3 is higher priority it can be serviced ahead of Req 0 as soon as the miss returns.

In principle you could worry about priority inversion, and insist that the priorities of Req 5 or Req 4 be passed down to the head of the queue if they are higher than the head of the queue. In practice, I think the only cases where we actually need to enforce ordering are requests from the same warp, so automatically already at the same priority.

I *think* also the differences in priority can come about because

- the requests are must be within the same kernel, but could actually be to different pages that are within the same superpage (ie a run of 4 or 8 or 16 16kB pages that will all be looked up at once, and installed as a set of validity bitmaps), so different pages with different types of content; but all serviced by the same miss?
- one of these page requests may be, eg a register access (highest priority) while another may be eg a Scratchpad access (medium priority) or a texture or ray tracing access (lowest priority)?



The above is my best analysis of what's going on, but I find it unsatisfactory.

We have good reason to believe (diagram from a patent!) that the “traditional” TLB sits between the GPU L2 and SLC, so sees low bandwidth and is probably far from these ordering concerns.

It seems reasonable (ie feasible, and optimal in terms of energy) that the rTLB translating graphics addresses into virtual addresses sits between GPU “L1” (ie pool of SRAM) and GPU L2, meaning that it also has fairly low bandwidth. Meaning that it's unclear quite why Apple thinks there is enough of a problem here that it needs to be solved in a more complicated way rather than just accepting the obvious, easier solution.

Maybe the issue is something like context switches? At a transition between threadblocks, or start of a new kernel, say, maybe there's an initial flurry of requests of every sort (registers, load/store, texture, ray tracing, etc) that all miss in the nearest rTLB, and during these brief periods there's an advantage to being able to maximally reorder requests?

unified memory?

It's obvious what Apple means by unified memory; Apple Silicon sees a single pool of DRAM living in a single physical address space.

And as far GPU/CPU interop is concerned, there's something of a unified address virtual address space, with all that implies for isolation, when it comes to transferring pages between the two, at the start and end of a kernel.

This means that, for example and (at least in principle) I can fill in a buffer of data on the CPU that starts at and fits in virtual page X, give that to Metal, and the GPU can, without much drama, load lines from that virtual page X through the SLC, into the L2, and ultimately into a GPU core.

Isn't that all we might want?

Not really! Suppose I want the GPU to engage in some sort of tree-walking exercise, so that on the CPU side I build up a tree by storing various pointers into the buffer. Then I expect, on the GPU side, each of my 32 lanes to be able to read an independent (virtual space) pointer value from this buffer, then be able to dereference that pointer and load from it. To work, this would require that each of the 32 lanes, for every load, must perform a TLB lookup. If my understanding of the various address spaces, as described above, is correct, this does not happen! The only place we translate an address from internal address space to physical address space is when we cross the L2/SLC boundary.

In principle this could still work; we could imagine this being like a VIVT system, with all the limitations that has (have to flush the cache on context switch, but we expect context switching on GPUs to be expensive; and aliasing is an issue, but we don't promise that virtual mapping games like shmem are going to work for the GPU).

So could Apple's GPU work (seamlessly interoperate with the CPU) on these sorts of complex pointer-based data structures, which to my mind is implied by "real" unified memory.

I'm told by people who actually program with Metal today that this is not possible. What's unclear is whether it's not possible because Metal still has to interoperate with AMD and Intel iGPUs (until 2026...), or because Apple Silicon does not allow it.

The same functionality is feasible through SW, simply by defining a large buffer to hold eg your tree, and defining a pointer not as a pointer directly but as an offset into that buffer. In C or C++ this could be wrapped in some syntactical sugar to make it slightly less painful, but would still be a hassle. A future Metal based on Swift could do better by simply being aware that certain "pointers" are to be shared between CPU and GPU, and doing this background transformation from pointer to buffer offset for you. Maybe Apple's choices in this respect will become more clear at the end of 2026?

Some Numbers; and Future Improvements?

At this point we can derive some value from Philip Turner's results in <https://github.com/philipturner/metal-benchmarks>

Most of his results should make immediate sense; the ones that don't make sense to me seem to reflect his analysis of the GPU which I don't understand, in particular I think he ignores the constraints of the register cache bandwidth and ascribes the limitation that imposes on other elements of the design.

What dimensions of improvement are possible?

Along with vector operations (ie operations across the entire set of 32 lanes) you may wish to execute scalar instructions (eg manipulating some sort of counter, or reducing across the set of lanes to a single value like an error which you then test). You can perform this sort of scalar manipulation by using (via predicates) one lane of the 32 lanes, but that seems like a waste. The alternative is to create something like a 33'rd lane, a "scalar" lane, and have "scalar" instructions execute only in that lane rather than across the 32 "vector" lanes. This, as I understand it, is what AMD does. Apple has the first part of this in that a thread has access to a set of up to 256 "uniform" registers which are scalar registers (ie a single value, not a set of 32 per-lane values); but as far as I can tell they don't have the second part, namely a dedicated scalar ALU.

A remaining prime inefficiency is branches, and in particular lanes that are unused for some operations depending on the predicate mask. I think this could be handled rather better. It's difficult to do this completely in hardware, but I think an ISA+compiler solution is completely feasible.

Instructions can be many bytes long, though usually this is not used, but I propose we exploit this.

Define a new instruction that looks like

```
both pN {if 0 op1 else op 2}
```

This encodes two operations in one instruction, one for the set of lanes that are masked, one for the set of lanes that are unmasked.

Sure, not every pair of `if/else` blocks can be fully fitted into this. Some will have more operations in one block than in the other; some will have operations in one block that are so complex they can't be paired (for instruction length, or semantics reasons) with an operation from the other block. But the goal is never perfection, it's just to capture 70% or more of the available benefit, and I think this proposal can do that with acceptable extra area and work in the instruction decoder and the compiler.

The one practical sticking point may be the register cache, which may not be able to supply the register bandwidth required.

However if lane-by-lane execution is somewhat decoupled (as was done for Volta) then much of the work to handle a "both" instruction will have to be done anyway.

There have been other ideas for how to deal with branch divergence. This is most obviously seen (ie the most well-known use case) is ray tracing where, even if we start a SIMD with rays pointing in essentially the same direction, as the rays propagate and reflect/refract/spawn new rays, more and more `if()` statements creep in and fewer and fewer rays are executing in the SIMD during any cycle. But ray tracing is a special case, which we'll deal with later; right now we care about generic branching.

An initial suggestion for how to deal with branching is given in (2007) <https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=94bead0f754d0d749ff05f59b9888d2ffadb42a> *Dynamic Warp Forma-*

tion and Scheduling for Efficient GPU Control Flow. This scheme can improve performance by about 20% across a set of general (non-graphics) benchmarks.

This sounds good, but there are at least two issues.

The first is that any scheme to coalesce across divergent branches requires a large pool of threads all running the same code. In other words, it really wants us to maintain a single large threadblock on each quadrant for as long as possible, giving up the advantages of interleaving different kernels. This seems intrinsic to the very idea of branch coalescing, and probably means such kernels should be statically tagged by the compiler to allow for optimal (non-interleaved) scheduling.

The second problem is that (implicitly, but it tends to work out OK), in the absence of branch coalescing, threads scheduled together tend to access memory together. Coalescing branches from different SIMDs within a threadblock can lead to diverging memory accesses by the resultant coalesced SIMD. This may well be a worse outcome than allowing divergent branches and a resultant less efficient execution! The paper above gives a few examples where this in fact happens, even taking in account that this is an early GPU paper and so the simulated GPU does not have a cache and is not optimizing scheduling to take memory reuse into account.

The followup paper (2011) <https://people.ece.ubc.ca/~aamodt/publications/papers/wwlfung.hpc.ca2011.pdf> *Thread Block Compaction for Efficient SIMD Control Flow* tries to improve the situation regarding memory divergence. The mechanism is more or less the same, but various details are tweaked, essentially to allow for fewer fully repacked SIMDs, but more frequent “total reconvergence” allowing for all lanes to be returned to their original SIMDs (thus, mostly, returning to memory locality within a SIMD). However it’s not clear to me that it’s good enough. What we’re seeing here is a repeat of a theme we keep seeing over and over again – energy should be our primary concern, moving data a long way costs a lot more energy than does most compute, and so naive worrying about the “inefficiency” of branch divergence may be a false economy. Perhaps we should be inverting the divergence problem, to try to rearrange SIMDs to optimize for coalescing memory accesses even at the expense of lanes that are less full?

Another way to look at this is to think of many recent NPUs (not ANE, but something like Intel, or Tesla’s Dojo <https://chipsandcheese.com/2022/09/01/hot-chips-34-teslas-dojo-microarchitecture/>).

These devices are something like a simple in-order 1-wide CPU that can also execute very wide (at least 512b) vector instructions. You can think of them as executing blocks of vector code that perform throughput work, connected by short runs of integer code that make global decisions (which block do we execute next? which algorithm to use next based on what we have calculated so far?)

It feels to me like a GPU based on these ideas could be a best of both worlds. A light, in-order, 1-wide CPU, using the uniform registers including something equivalent to a uniform set of flags, is basically the equivalent of the scalar line I am suggesting be added to the GPU. Something like this is probably also valuable for the ANE (you will get my point once you read the ANE volume); it *may* already be present in each ANE core, just Apple tells us nothing about it in the patents?

The current Apple scheme is that something like what I am calling the scalar code, the blocks that tie together the blocks of serious throughput code, executes in some form on the companion processor

of the GPU or ANE; but that seems sub-optimal – too far away from the decisions and not actually programmable by the developer (GPU or neural net designer), more a set of pre-set functionalities selected by the driver or compiler as appropriate. Better than having no such escape clause, but sub-optimal given how lightweight you can design the target CPU/lane.

locality

What else can we do?

One suggestion that seems especially relevant to Apple is (2018) https://people.inf.ethz.ch/omutlu/publications/LocalityDescriptor-Cross-Layer-GPU-Data-Locality-Abstraction_isca18.pdf *The Locality Descriptor: A Holistic Cross-Layer Abstraction to Express Data Locality in GPUs*. This suggests annotating GPU memory allocations with annotations that describe something of the how the data is used within a threadblock and across threadblocks. This could be done within Metal by using the `[[keyword]]` syntax, and could be used to better inform things like the current algorithms for how threadblocks are allocated within a grid, and how they are co-scheduled.

Implementing different types of load/stores could be done using the same sort of hint bits scheme as we use for the operand cache.

But another alternative (which might be more flexible) is to give each load and store instruction say a 4 bit field. This field, when 0, means nothing special, otherwise it indexes into a per-kernel or per-process 15-element “data streamID” table.

Elements of this table could hold a DSID and various flags (bypass cache, cache normally, pin weakly, pin strongly, etc) for L1, L2, and SLC. This would also fix some of the limitations in the current DSID scheme and how some functionality (like cache purging) is of limited benefit because there isn’t, right now, an easy way to indicate except at a very broad level the specific lines to be, eg purged/dropped.

One way to think of the paper is that right now the only real info available for creating, and then scheduling, threadblocks is an assumption that the grid structure maps in some way to the data access patterns. Even to the extent that this is true (a 3D grid is accessing a 3D data structure in a local fashion) the mapping of the grid onto the data structure (eg the ordering of the axes) is not obvious.

A separate way to indicate the way the most important data structure(s) are accessed can only help the threadblock construction and scheduling.

The paper is also interesting for some non-obvious insights into how GPU memory access plays out. For example, across a variety of workloads, it investigates different ways of building a threadblock within a grid, something Apple also tries to do carefully.

What they find is that while doing this optimally does result in substantially lower memory traffic (good, lower energy) it does not result in much performance improvement! The reason is that the optimal threadblock geometry and scheduling tends to result in multiple threadblocks all requesting the same cache line at essentially the same time, so that most of the active threads stall simultaneously and there’s no work for the GPU to do till data returns.

This is the sort of not completely trivial insight that suggests there is still a lot of scope for GPU improve-

ment, for example in using these sorts of annotations not just in some obvious ways (scheduling, as suggested in the paper; or to suggest DSIDs and how they are used) but also to inform various types of GPU-specific prefetching (eg only prefetch on very high confidence; informed by the behavior of earlier threadblocks in a grid; and perhaps only to L2, not L1).

An alternative strategy is described in (2021) <https://www.mdpi.com/2072-666X/12/10/1262> *Locality-Based Cache Management and Warp Scheduling for Reducing Cache Contention in GPU*. This paper suggests that, rather than manual/compiler annotations to each data structure, the GPU tracks the behavior of each load. A particular load (defined by a PC) may be repeated within a loop, and will definitely be repeated for every SIMD within a threadblock, and across all the threadblocks in a kernel. The idea is that, using the first few behaviors of the line loaded by that PC, we classify the load by how it tends to reuse data, and use that to inform subsequent loads.

The most significant data points in the paper are that, for the benchmarks chosen, performance of a naive GPU, using GTO scheduling, is remarkably insensitive to L1D size larger than 16kB, or to cache associativity. In other words you get most of the value you can get from GTO scheduling reuse with a small L1D, and you need something fancier if you believe there is some sort of long-distance data reuse.

Of the various ways they attempted to utilize the classification provided by their table of load behaviors, by far the most significant (20% speedup) was to bypass streaming data, avoiding the cache and placing values directly in registers, and so not wasting cache space (and latency, and energy) storing lines that will never be reused.

This analysis confirms much of Apple's design (eg small L1D, attempts to use DSID to pin data in the SLC for reuse), but suggests that there is scope for improvement by Apple in implementing cache bypassing, either by automatic detection or by DSID or similar tagging of data streams. (Possibly Apple already do this, but I found no reference to it anywhere. And bypassing is a powerful idea that can appropriately be implemented not just at L1, but also L2 or even SLC, given how much GPU data is non-temporal).

These all seem like a lot of ideas, but ultimately they come down to

- how should we schedule kernels, threadblocks, and instructions to optimize cache reuse (and in the face of other desiderata)?

We can come up with adequate heuristics, but we can do better if we have telemetry from the cache/load/store subsystems, and/or if we have annotation data that tells us of the links between how a thread grid is laid out and how the data it uses is laid out.

Relevant decisions include

- + interleaving different kernels on the same core vs separating them to different cores;
and the same decision but split across different GPUs (or halves of an Ultra)
- + increasing the number of threads running on a quadrant/core (lots of scheduling options) vs
limiting that number (better L1D reuse).

One thing these papers show is that there is no optimal solution to these scheduling questions; different workload have different best answers. And so going forward we will probably see even more telem-

try (describing cache reuse, bandwidth to L2 and DRAM, and execution unit activity levels) passed upward, so that scheduling policies are changed as appropriate.

For example the academic literature is quite clear that there are cases where scheduling should limit the number of concurrently active SIMDs on a core below the maximum possible to prevent L1D cache thrashing, even if it results in occasional (or frequent!) execution bubbles. (This is not a small effect. For some classes of benchmarks, the optimal number of concurrent SIMDs is a quarter to an eighth of the maximum the GPU core supports, and the consequent speedup is 1.5 \times to 4 \times !)

Soon below we will see an Apple patent that addresses this same issue of cache thrashing. The Apple patent talks about this mainly in the context of excessive register usage, or excessive L0 I-cache thrashing; but one hopes that the implementation is generic enough that it catches all forms of cache thrashing.

OoO

What about OoO execution? Superficially this might seem unhelpful or irrelevant to a GPU. But let's think it though.

In the CPU world why do you care about OoO? The compiler can rearrange ("schedule") code to work around the most obvious dependencies, but there are two big limitations.

The first is unpredictable latencies, especially loads. For the most part GPUs can work around that by shifting threads.

The second is that the compiler cannot rearrange instructions across branches. This is where there may be some value for a GPU.

Imagine code like

```
work A
compare
conditionally work B
```

An in-order CPU (or GPU) cannot execute the work B items interleaved with the work A items.

Even if it's possible for the compiler to move the `compare` instruction up before the `work A`, the details of how the predication stack works still don't allow interleaving code A and code B. (On a CPU with explicit predication instructions such interleaving *might* be possible, depending on the code details.)

If our GPU supports a very small degree of OoO execution, within just a window of 3 to 6 instructions (essentially like the first few PPC designs, eg the 601 or 603) then we can perform some degree of instruction rearrangement at these boundaries and gain both some per-thread latency and (especially if we have implemented some superscalarity) some throughput.

We can also, if we want,

- add register renaming. This avoids some delays in loops, where subsequent loop iterations have to wait for earlier iterations to writeback their results. This can be avoided by loop unrolling, at the cost of using more registers and more instructions. We already have a lookup table for register names

(required to support packing different threadblocks using different numbers of registers, onto the core; and then extended via the M3 register virtualization). BUT that existing lookup table is designed for setting those register names statically at the start of each threadblock, not allocating a destination register name every cycle. Is this worth doing? Hmm...

- engage in some degree of loads moved to before stores. You don't want to do this speculatively, but you can often test the store address vs the load address and see that things are OK, in a way that the compiler cannot. This may be worth doing.

A suggestion covering these sorts of ideas is presented in (2021) https://drive.google.com/file/d/1c_IfJZxoSCKTJ37yYxyW1_I7qYIEKVL/view *Repurposing GPU Microarchitectures with Light-Weight Out-Of-Order Execution*.

FP64 support

At least for now, Apple's expectation is probably that if you have FP64 code, you're best off running it on AMX, and if you want even more performance, splitting it into parts that run on AMX and parts that run on NEON. Some people have looked to the performance of "interesting" FP64 code run on Apple GPUs on the assumption that even the slower FP64 on a Max might be better than AMX, but right now that appears not to be the case. The discussion in this thread <https://forums.macrumors.com/thread-s/apple-silicon-in-sciences.2374458/page-3?post=31890787#post-31890787> is all over the place, but occasional of the posts and numbers refer to FP64 on the GPU.

There are different a few different ways to fake a 64 bit number using 32bits, depending on whether you only care about extending the precision of the mantissa, or whether you also care about having the full, wider, exponent range. These have names like e8m48 (8bit exponent, like FP32, but 48bit mantissa) vs e11m48 (11 exponent bits like FP64, but still only 48 mantissa bits).

Full IEEE FP64 would e11m53, which is a lot more work than either of the two m48 options, so generally ignored for practical science on a consumer GPU. On Apple (and other consumer) GPU's e8m48 seems to be about 20x slowdown compared to FP32, e11m48 seems to be about a 100x slowdown.

Apple could (if they saw reasons to do so) improve this situation. One possibility is to still do FP64-like math in the ways described above, as manipulation of mantissas and exponents, but with a few helper instructions to make the process more efficient. Another possibility is actually provide some support in the FP32 unit for FP64 math. There is actually a patent for this (2013) <https://patents.google.com/patent/US20150058389A1> *Extended multiply*, by two prominent names on the Apple GPU team, discussing how to handle either FP32 or FP64 on FP16 hardware. But this seems to be an idea that (at least so far) has gone nowhere.

superscalar

Next we could provide instructions for FP16[2] pairs. These could either execute one FP16 on the FP16 hardware, one on the FP32 hardware; or we could modify the FP32 hardware to handle FP16[2] pairs.

These two ideas would essentially match nVidia and AMD, and probably would not stress the register cache too much.

Can we do better?

Ultimately we have 4×32 lanes of execution machinery. But each lane has multiple (essentially independent hardware) execution units: two FP32 and an FP16 pipeline, an INT32 ALU, and a load store unit, among others. Wouldn't it be nice to go superscalar and send instructions down multiple of these execution units each cycle?

The obvious idea is that once four-wide issue is basically perfected, the next step (requiring, again, ramping up the operand bandwidth) will be to provide a sort of constrained dual-issue within each sub-pipeline, something like the Pentium's U and V pipelines.

We already described above, in the operand cache section, how the multiple mini-frontends currently allow between the selection of a dispatchable instruction from four (probably six in the M1 and M2) possible mini-frontends. That multi-instruction dispatch only requires that independent execution units be available, it is not limited by dependencies between instructions.

U and V dispatch is more sophisticated, now sending two instructions down each mini-frontends, and having to test possible dependencies between them. On the other hand, it does reduce latency within a single thread, and provide some execution parallelism even when not all six (or more) possible active threads are actually active.

The optimal model might be to implement 2-wide dispatch first between distinct mini-frontends and then, when that is working, later add on as much as is easily implementable of 2-wide dispatch within a single mini-frontend.

That all sounds good BUT there is the problem that all these ideas would overly stress the register cache, which appears to be already operating at about peak throughput, and widening that is not free. Maybe the cost is worth it (in that it provides more bang for the area than the alternatives of adding more cores)? Or maybe using heuristics, we can do an adequate job of splitting the cache into more or less independently accessed caches (eg one cache that's holding int values, one that's holding FP16 values, and one that's holding FP32 values)? As always, knowing what's really optimal relies on information I don't have :-(

Also we need to optimize for what is the most serious problem. It's true that FMA units are large, so that it makes sense to try to optimize for their use 100% of the time; but it may also be true that, all things considered, the load/store machinery takes up even more area, it's just less obvious because it's distributed over multiple different pieces (queues and sorting logic, TLB, buffers, etc); and so we should work even harder to make sure that this LS machinery is 100% occupied?

In which case it would be more important to try to figure out/fix the bottlenecks in load/store processing, and leave the FMAs to take care of themselves...

Certainly there is a long history (which I think is still true) that load/store is the hardest part of any design, and mostly the real bottleneck on real code. It's fun to fantasize about adding more execution; but it's more important to make load/store work better...

(On the CPU side this is more obvious, Apple GPU has the advantage of not just the equivalent of load pair, but the equivalent of "load vector", so a single load can pull in a lot of material for subsequent

execution.)

So we can imagine a trajectory for Apple over the next few generations that can squeeze quite a bit more performance out of their design without requiring *that* much more hardware. (A lot more work on the operand cache, slightly more work on the sub-core scheduler and a new within-sub-core scheduler, and improved L1D bandwidth; but not requiring much more in the way of ALU's except for bolting a scalar ALU alongside the SIMD ALU.)

Interestingly, the idea of dual (or even triple) dispatch to a lane is not that new. This article <http://www.anandtech.com/comments/7793/imaginations-powervr-rogue-architecture-exposed> describes the PowerVR 6 series, as used in the Apple A8. It's interesting to read in terms of what I have described, but the point I want to draw attention to is

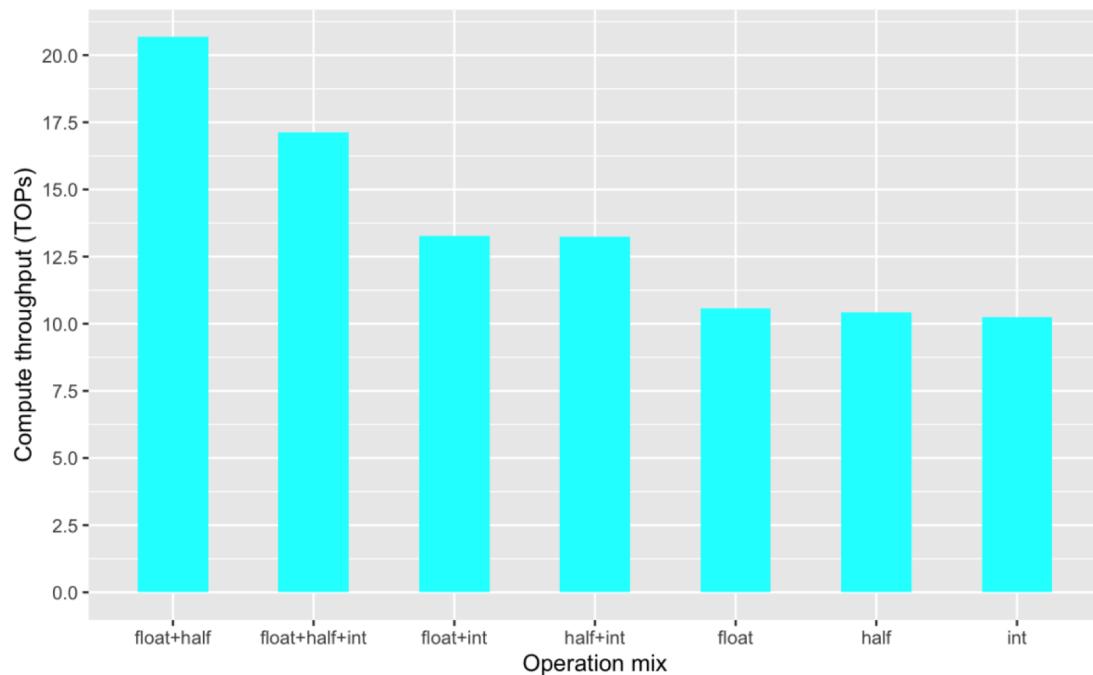
- the PowerVR scheduler scheduled two (or perhaps even up to four) simultaneous instructions to a set of 16 lanes
- one aspect of how they could do this is that they ran the scheduler at half speed (more time to figure what to do) then submitted each instruction (or instruction pair or triple or whatever) twice to each lane, so in a sense there were 32 "logical" pipelines implemented as even and odd cycles on 16 single physical pipelines! We have seen that nVidia has also used this idea at different times.

Now that Apple have abstracted away both register pressure and threadblock local memory, new types of design points open up, beyond the 2020 consensus that nV, AMD and Apple all gravitated towards.

For example imagine splitting the current design (four quadrants) into six hexants, but each hextant is only 16 not 32 wide, and executes instructions over two cycles. Each mini-front-end now has two cycles to decide on the optimal next instruction(s), and more options open up for scheduling two instructions from a single thread, so a first stab at UV pairing.

update

It appears that t least some of the above suggestion has been implemented as of the M3. From "leman" <https://forums.macrumors.com/threads/m4-chip-generation-speculation-megathread-merged.2393843/page-22?post=33149109#post-33149109> we get the following graph:



What this tells us is that we can now (at least M3, maybe M2?) dual-issue FP32 and FP16 instructions (and we have enough register cache bandwidth) to get both those units executing simultaneously. We also see that there is no such dual issue (of course, how could there be?) for a stream of purely INT, purely FP16, or purely FP32 values. INT seems to sit partway, clearly some additional issuing is possible, but not every cycle, perhaps because of register cache bandwidth? With dual issue there's also a little more reason to add an additional scalar data lane...

a superscalar patent update (2022)

A few days after benchmark confirmation above, I found the following very recently published patent (2022) <https://patents.google.com/patent/US11954492B1> *Fence enforcement techniques based on stall*

characteristics.

Before getting to the patent details, let's note that it's an update to the "Multi-channel data path circuitry" which introduced the idea of the mini-front-end, which the patent calls a channel. So let's remind ourselves of the idea.

The original GPU can hold, in one core, something like up to $4 \times 24 = 96$ "active" warps, ie threads that have been allocated registers. At any given time probably half or more of these are in a "suspended" state, executing a load/store clause, and waiting for memory activity. So a particular quadrant generally has a much smaller pool of warps from which it can actually execute datapath (ie FP and INT) instructions. From this smaller pool it chooses about 6 or so (based on various things including how long a given thread has gone without executing, and trying to balance threads that, via compiler hints, make more or less use of specialized pieces of hardware like the Texture Unit). From the 6 or so active warps in any given cycle it chooses one to execute next, as best it can.

Limitations on executing next may include that instructions are not available (miss in the L0 [clause] cache which will have to route to the L1 cache and maybe higher), or that register values are not present in the operand cache, or that the instruction depends on an earlier instruction on this thread, or the target execution unit is active on some other task, or more exotic issues like fences and memory barriers.

On the one hand, if we had a larger pool of active threads, we could maybe do a better task of always finding a thread ready to go with no delay. On the other hand, this is already a fair amount of detail we have to check for the next instruction of each active thread, some of which depends on earlier activity (or we simply may not know at this point, like operand cache availability).

So can we improve the situation?

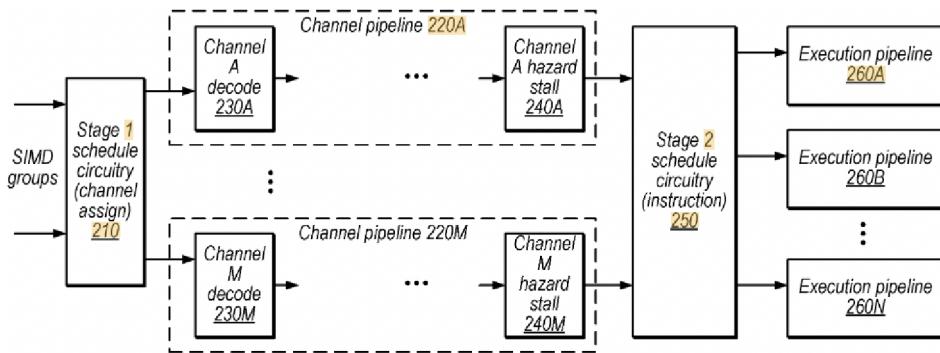
One way to improve things is to enlarge the pool of potential active threads. We do this via the new (M3 and A17) resource allocation scheme that provides a single pool of SRAM for all near resources (Scratchpad, registers, ray tracing, etc), using virtual memory like techniques to rename pieces of the SRAM to whatever resource is required, and with the ability to spill not immediately required resources from the SRAM to L2. This allows us to pack many more threads into a core, and, if appropriate, to reallocate the resources of threads that are otherwise blocked.

But that's no good unless we can accurately schedule all these warps, and that's where the channels (mini-front-ends) come in. By providing multiple lightweight instruction processors, we can let each channel begin accessing one of multiple threads, and allow that thread to pause partway down the channel if there is some minor problem (like waiting for the operand cache, or for the result of an earlier calculation). If we have, say, four of these channels feeding a single instruction issue, then hopefully there will always be, at the issue point of the four pipelines, at least one instruction ready to go.

So far so familiar. This is drawn below, but there's a nice twist to the diagram. Now instead of abstracting what happens after stage 2 as a single pipeline we split this into multiple pipelines. For

example, if these are channels executing Datapath clauses, then the three obvious pipelines are FP32, FP16, and INT. There's probably also an additional Special Function pipeline (perhaps 8-wide rather than 32-wide) and you could imagine more such (perhaps a Fast Special Function pipeline that's 8-wide handling Sqrt and Reciprocal, and a Slow Special Function pipeline that's 4-wide handling transcendental functions?)

Given multiple execution pipelines, we can now strive to have something like say six active channels feeding Stage 2, which then chooses two instructions ready to feed into two different pipelines. (And of course we'd prefer in a perfect world to be able to feed five instructions into five pipelines, but other constraints, most obviously Operand Cache bandwidth, will have to be handled before that's feasible.) Our benchmark shows that we're already making some use of this idea in the M3, probably just at the level of issuing two instructions per cycle.



The patent, then addresses the on-going question of how best to execute both Stage 1 and Stage 2 scheduling.

We accept that some threads will block in a channel for a few cycles (that's what the channel is for!) but we want to limit it as much as possible, so that Stage 2 has as many valid channels as possible, providing at least two instruction options. So how can we improve Stage 1?

We start with cache hints and age (ideally we'll always have a balance of threads that use FP32, FP16, and INT flowing into the channels; and all other things being equal we'll prefer to Stage 1 schedule threads that have not been executed for some time).

I would have thought that at this stage (Stage 1) it should also be known whether a remaining instruction is available in the L0 clause cache, or we have an l-cache miss; but the patent is somewhat vague on this point. What is clear is that we can't know (because this is only learned at Decode) what operands the instruction uses, so it is definitely possible that we may have to block because of an Operand Cache miss. For the same reason we also can't perfectly schedule at Stage 1 based on balancing FP32, FP16 and INT; the best we can do is use compiler hints to selectively balance threads that are

hinted as being especially heavy in FP32, FP16, or INT.

We will also need to check, as we flow down the channel, whether there are any relevant fences/barriers, or hazards (most obviously use of a register calculated by a previous instruction).

Then, at Stage 2, we may have multiple instructions executable, and even multiple instructions targeting say FP32. So there's also a non-trivial decision to be made as to which of the various candidates gets a given pipeline. Or, if we have INT, FP16, and FP32 ready to go but can only fire up two pipelines, which two pipelines?

So with all these concerns, what's Stage 1 to do?

The basic idea is that we try to schedule threads by age, but

- we compare back-pressure from the execution units (which execution units are being worked harder and less hard) against the compiler hints
- based on these comparisons we tweak the real ages of threads to "scheduling" ages, to try to ensure that threads targeting less stressed execution units are preferentially scheduled.

Next we move into the channel.

New, relative to the first mini front end patent, is that we can now *de-activate* threads that stall in a channel. If a thread starts down a channel but we believe that its stall will take a long time, we can throw it out, back to Stage 1 scheduling (presumably with some sort of attached "sleep value" or "wake condition" so that it isn't immediately re-scheduled by Stage 1!

The most likely cause of this (as far as I can tell) is if the thread has had some of its registers swapped from SRAM out to L2, so that a test against the Operand Cache shows that bringing in the registers will take more than just a few cycles accessing the local SRAM.

The primary point of the patent is to weave all these ideas together specifically regarding fences and dependencies. If a thread cannot proceed until some dependency is cleared, it makes a difference if the dependency is likely to be resolved soon (in which case we let the thread pause in the channel) or if the dependency may depend on thing happening out in L2 or even SRAM and DRAM (in which case we want to de-activate the thread from the channel).

So the star of the show is below, showing how earlier execution (state of instructions that affect a fence/dependency) is communicated to a manager which in turn lets the channel know if certain in-progress threads should be de-activated.

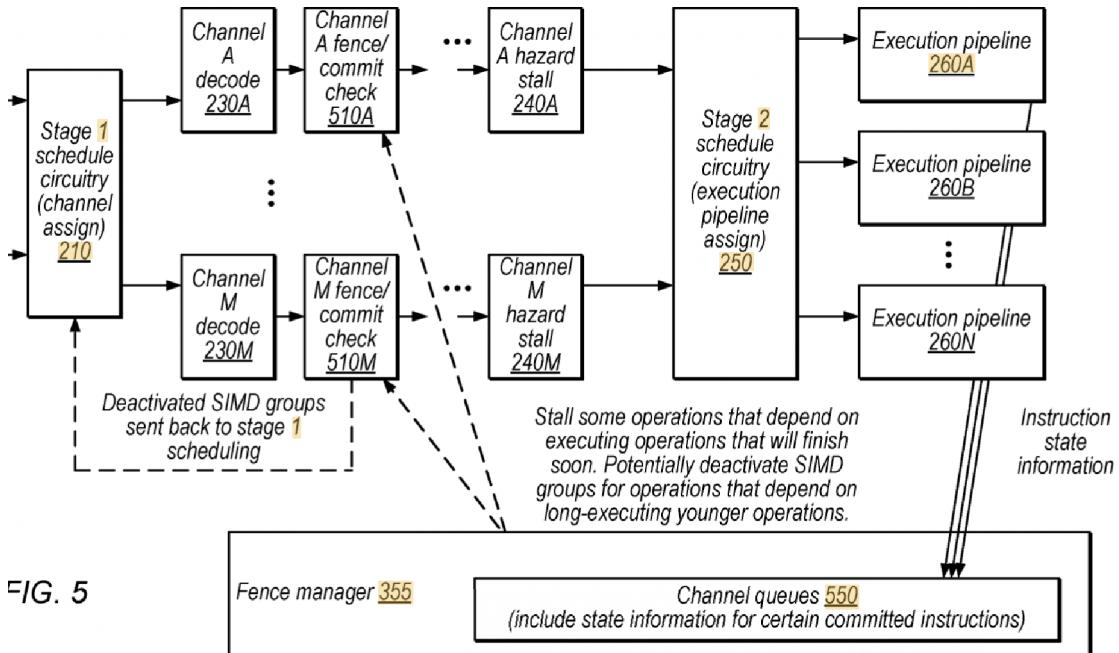


FIG. 5

The system categorizes expected delays as short, medium, long, and extreme.

Short delays (maybe up to five 8 cycles or so) are based on waiting for a result from an execution unit, and the channel just stalls, allowing other channels to provide instructions.

Medium delays may involve nearby (level 0) cache access. The patent describes them as stalling in the same way as short delays, but maybe there are power implications, like the channel is powered down and then retried again after ten or twenty cycles?

Long delays are things like worst case on-core (L1) cache access (and maybe some fast L2 accesses?). They result in thread de-activation.

On extreme delays (so most, maybe all, L2 accesses and further) the thread is “context switched out”. This is a somewhat confusing term given the new memory design. Traditionally, what limited how many threads we could have active was either we ran out of registers or (less likely, but possible if dealing with small threadgroups) we ran out of Scratchpad space. Recall that the new M3 GPU design virtualizes these resources so that there is no real cap on the number of registers (or Scratchpad space), we can just keep allocating these as we like, and the only consequence will be that more and

more of this state can no longer fit into the local SRAM and has to be spilled out to SRAM (like the L2) that's further away. So we have a somewhat different situation from before, where we have to balance the number of threads we allow to be active with the amount of thrashing we see happening against the most local SRAM in which registers are placed.

In many situations we may not need to actually context switch a thread, what we could do is something more lightweight. For example we could mark all the lines in this local SRAM that are being used by a thread that has missed to L2 as least recently used, then just let them be aged out naturally over time.

Even if we do wish something like a “real” context switch, we can do so by simply flushing those lines from the local SRAM. In both cases, when we wish to context switch the thread back in, we can simply start using the thread again, and relying on the cache miss mechanism to pull in the thread data (registers and so on) on demand and spread out over time. As long as we have enough other threads ready to execute in each channel, both sides of the context switch will cost us less time because they will simply involve background work by the caching mechanism, while the GPU core itself (and specifically the load-store units) are doing other work.

If we compare this to the CPU, the CPU, as we have seen, makes aggressive efforts to ensure that blocked instructions (for example Replays) are made executable as soon as possible with cycle-accuracy, through the provision of many wires to wake up individual sleeping instructions of individual specific events. The GPU seems much more relaxed about this, willing to just delay things that are currently blocked by some amount that's hopefully enough to ensure no blocking the next time round. Of course the difference is that the GPU has many more threads from which to schedule, so losing one or ten or a hundred cycles (as long as those cycles are usefully occupied by some other thread) is worth the saving in area and energy.

On thing that's especially confusing about this patent is that it frequently suggests that load/stores and datapath instructions are co-scheduled. If we believe the clause model (and recent benchmarks seem to suggest that there's still some expense in switching from one type of instruction to another, which suggests a clause-like design) then my model would be that different channels are associated with different clause types, and thus different L0 caches and execution units.

But maybe that's incorrect? Maybe there's a common pool of channels that handles all clauses, and so does not know which clause cache it will be accessing until it starts decoding the info for the given thread? Which might explain how I-cache miss can occur within a channel?

Maybe it's more area efficient to be clause agnostic (a single IL0 clause cached, highly banked to allow multiple independent accesses per cycle) feeding a common pool of channels that handle all dependencies, and only splitting to per-clause execution (load/store vs datapath vs texture) at L2 scheduling?

This seems to be behind one technical detail of I-miss handling. Suppose, at the start of a channel, we go looking for the instruction and don't find it in the IL0 cache. We deactivate the thread while we go looking for it in IL1, L2, or wherever. But the following can occur, that by the time we restart the thread it again misses in IL0! In which case we force a stall for the thread, not a deactivate. The patent

describes some detail to this, and to my eyes it looks like clause-cache behavior. We install the clause for a thread in the (small, and now, apparently, shared) clause cache, but the vagaries of Stage 1 scheduling mean that by the time we re-activate the thread the clause is no longer present. However it's almost certain that the instructions are still present in the (much larger) IL1 cache, so there's probably not too much harm in simply stalling the thread this second time, till the clause is constructed from the IL1 cache.

Two related patents are (2022) <https://patents.google.com/patent/US20240095176A1> *Preemption Techniques for Memory-Backed Registers* which discusses in a little more detail the issue of context-switching threads simply by moving registers from local SRAM to SRAM that's slightly further away; and (2022) <https://patents.google.com/patent/US11947462B1> *Cache footprint management*, which discusses the tracking of local SRAM to detect when some sort of thrashing is occurring, in response to which we should reduce the number of threads being actively scheduled. What's tracked is the sort of thing you would expect, like the rate of misses in the local SRAM. But that's not the full story! Of course you can limit cache thrashing by executing one thread at a time; in which case the register SRAM never misses, but the execution units are frequently idle. So you also need to track the execution unit duty factor. It's tricky to even define exactly when we are thrashing – some combination of both frequent misses by the register SRAM and level of activity of the execution units, sure, but exactly what combination?

Without giving many details, the patents suggests, as additional non-obvious points, that

- both L0 and L1 cache (L0 is presumably local per-quadrant SRAM that mainly holds registers, and L1 is presumably a per-core more traditional cache, into which register overflow) are tracked
- at least for L0 in addition to tracking cache line turnover, proxies like the number of modified lines are also tracked, because these tend to indicate a likely upcoming problem faster than more obvious metrics, and can allow thrashing to be prevented (ie by pausing a thread from scheduling) rather than having to react after the fact (ie once a whole lot of excess cache miss traffic has been detected).

Even with the decision as to what constitutes thrashing sorted out, the decision as to what thread(s) to stop scheduling if we are thrashing is far from obvious. Let me throw out some possible criteria:

- the oldest thread (because it has had its chance, fairest to give someone else some time)
- the newest thread (it has accumulated the least state moved into local SRAM, so moving it out will use the least energy)
- the thread using the most local SRAM (again, fairness)
- the thread using the least local SRAM (again, energy)
- the thread that will have the least effect of execution unit throughput. In other words, if most of the threads are mostly doing FP32 work, then de-activate one of them, not a thread that is mostly doing FP16 or INT work
- make sure the thread being deactivated is NOT what is blocking any sort of fence or barrier, because we don't want to slow down multiple other threads with this one deactivation.
- can we maybe look ahead in the instruction stream at least 10 instructions or so (eg compiler hints) to know if the stream will be encountering any sort of fence or barrier (or will be ending soon, or will soon

naturally deactivate by performing a long latency task like a memory access)? If so, keep it around and let it naturally deactivate?

etc

And of course there is the reverse problem as well – if space opens up to schedule a new thread, which one should be chosen?

My guess is that over the next few years the details of how to handle this GPU cache thrashing will be reworked a few times to incorporate more and more of the above issues.

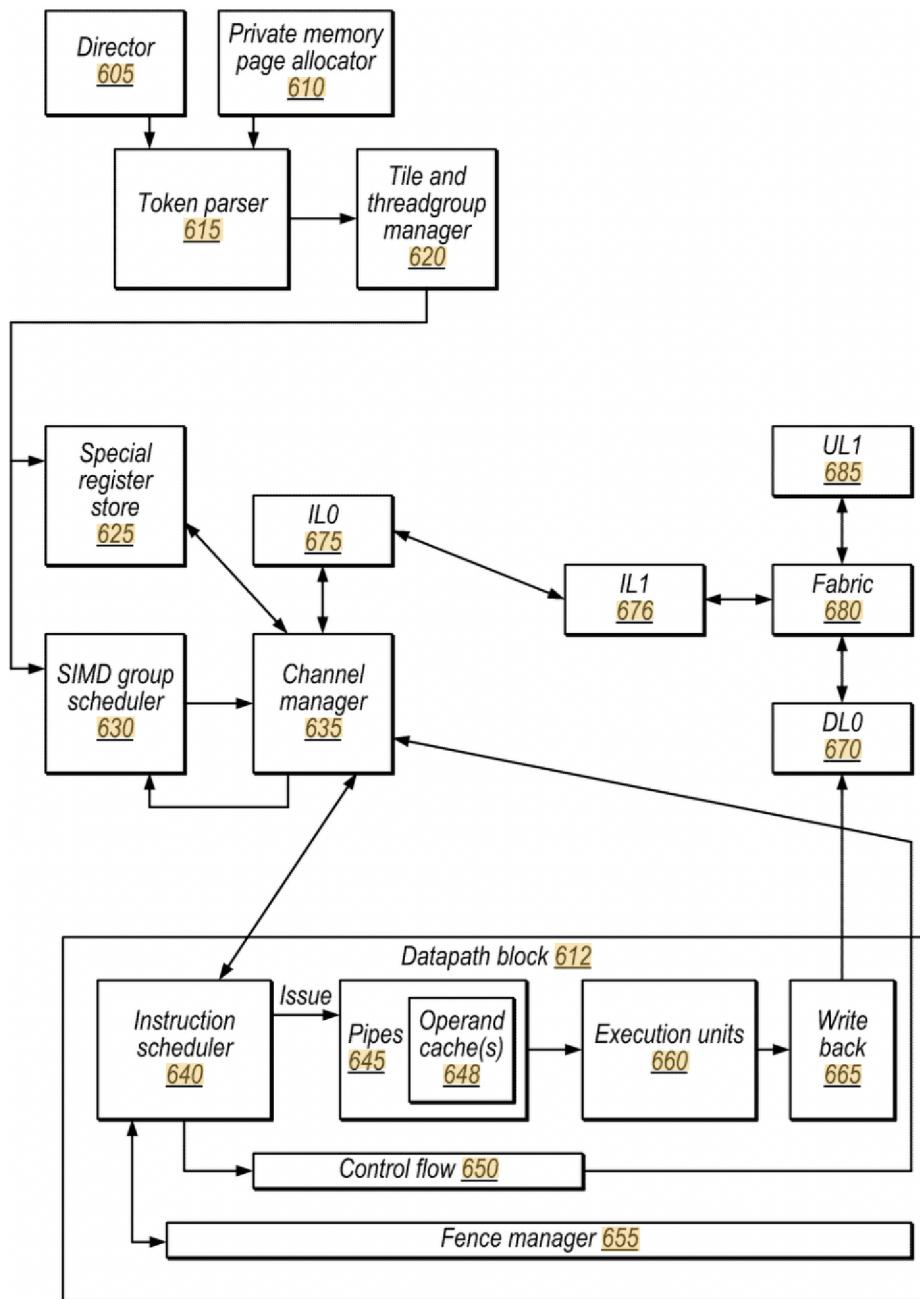
With (2023) <https://patents.google.com/patent/US20240289282A1> *Cache Control to Preserve Register Data* we see the next step in attempting to limit thrashing in this new, virtualized registers, model. The problem to be solved in this case is that we don't want instruction scheduling to issue an instruction (assuming the required registers are available in local SRAM) only to have the SRAM management system (ie “the SRAM caching system”, which is operating independently) decide this would be a good time to swap a lines containing these registers out to L2! On the other hand, a perfect solution to this problem might involve a lot of back and forth communication.

So the good-enough solution chosen is that when an instruction is scheduled, the relevant line in the SRAM is locked, which prevents the line from being swapped out. The trickiness in such a scheme is when do we then unlock the line? For now, at least, Apple punts on this. Lines are locked as required, and locks keep piling up until eventually a “reset event” occurs, which wipes all the locks. Reset events are things like a context switch (eg we give up on being able to make forward progress with this thread because we are waiting on RAM, so we clear all the locks for this thread's assets and let them page out if necessary, while we handle some other threads).

The patent includes this diagram which is helpful as an overview of the entire GPU system now.

My guess is that

- Director and Private memory page allocator (the largest scale kernel processing) happen in the GPU companion core (ie the ARM Apple Chinook core associated with the GPU)
- Token Parser, Tile and threadgroup manager, and UL1 (Unified L1 cache) happen at the GPU Core level
- Special Register store, IL0, DL0, SIMD Group Scheduler, Channel manager, and maybe maybe IL1 happen at the Core Quadrant level
- I think Channels are what I have been calling a mini-front-end, so some stuff like Instruction Scheduler/Issue is kinda split over a few channels, choosing a Channel that currently has a runnable instruction to route to the Operand cache and Execution units.



Texture Issues

I'm mostly not interested in deep graphics details, but there are a few unexpected issues related to texture mapping.

Let's think about what texture mapping entails, then we can compare the evolution of our conceptual model with Apple's evolution. In everything below I'm just going to sketch the outlines of the ideas; there's plenty of internet material on texture mapping if any of this is unfamiliar to you.

(Also, as a technical point, most of the time you can sample from 1D, 2D, or 3D textures but I'll describe everything in terms of 2D textures as the most common case.)

The simplest possible version of a separate Texture Processing Unit might be something like:

- the shader core, for each pixel (x, y) to be drawn, translates that (x, y) , in world space, to an address (u, v) on the triangle being drawn and passes the (u, v) to the TPU which is responsible for loading the texture (along with some sort of caching system), mapping the floating (u, v) co-ordinates to some closest match address offset in the texture, and handling any pixel format conversion that may be required.

Now let's add functionality one step at a time.

- as already mentioned, how do we handle (u, v) addresses that would map outside the texture? One option is to call this an error, but it's frequently useful (and saves memory) to handle such addresses by clamping, or wrap-around, or mirror-ing. We could make the tests of (u, v) against bounds, and then do the address clamp/wrap/whatever on the shader core, or we could delegate that to the TPU.

- mapping (u, v) to a single closest point in the texture gives rise to very visible aliasing artifacts, so we will want to map the (u, v) to say the nearest four points and linearly interpolate the color value at (u, v) from those four neighboring points. You could do this in the shader code, but again, why not delegate it to the TPU.

- if the triangle is far from the camera then a single screen pixel corresponds to a large region of the texture, and the color of the screen pixel should be the average color of that large region of the texture. The easiest and most performant way to achieve this is via MIP-Mapping, ie represent the texture now as a base texture, a $\frac{1}{2} \times \frac{1}{2}$ y texture, and so-on; ie a pyramid of textures, each one successively half the x and y dimensions of the previous level; then sample as before but from the appropriate level of detail, so a triangle that's far from the eye uses a texture that has already filtered down many of the base texture pixels onto a single reduced resolution pixel. Now the question is, how do we decide which LoD (level of detail) to sample from, and, once again, does the shader core do the work to decide that or move as much of it as possible to the TPU.

- we can improve quality even further by using so-called *trilinear* filtering where we don't just calculate

a bilinearly interpolated value at (u, v) from the best-match LoD in our texture-pyramid, instead we do this twice with the two LoDs that bound the optimal LoD we would like to use (eg an LoD that corresponds to, say, a shrink of 5.7) and then linearly interpolate between those values. Once again, ideally delegated to the TPU.

- textures are large, so it would be nice if we could store them compressed in RAM and have the TPU behind the scenes just deal with decompressing them, and caching the decompressed blocks in a texture L1 cache for as long as possible.

- we still aren't done. The work described above limits aliasing "in texture space", that is it limits blinking pixels and discontinuities that keep appearing and disappearing as say a triangle changes its orientation and so samples different points in texture space.

But this does not help with aliasing "in screen space". In other words, imagine a thin triangle that's moved from one column of vertical pixels to the next row. There will be a point in time where the thin triangle (ie think of a vertical line) covers one column of screen pixels and we draw each of those pixels very nicely and accurately using all the techniques described above. Now the line moves by half a pixel, so that it's supposed to be covering the original screen column and the next screen column. If we draw both columns in the same color, then the line keeps popping from thin to thick to thin to thick as it moves horizontally! What we want is to fill each screen pixel with a fractional brightness equivalent to the fraction of the triangle/line that covers the pixel, so that when the line equally covered two columns, each column was half as bright, and visually it still looked essentially like the same line emitting the same amount of light.

This is the task of MSAA, and the easiest way to achieve it is instead of just generating one (u, v) address from the (x, y) of the screen pixel, we generate say four (u, v) addresses corresponding to four points in the screen pixel, test each point for whether it does or does not lie within the triangle (since the triangle may only partially cover the pixel) and then submit the covered (u, v) addresses to the TPU.

There's then a whole drama of exactly where these four points should be placed in the screen pixel. The easiest option is some sort of fixed pattern, but fixed patterns tend to generate more visible artifacts, so another alternative is a random pattern, ie each screen pixel we generate four random points within the screen pixel and send those to the TPU. And once again the obvious question is how much of this work (generate four [or eight or whatever] points within a screen pixel, test them against triangle bounds, then sample at each point and sum) do I have to do in the shader vs how much can I offload to the TPU.

Now let's see how Apple has tackled various of these challenges.

(2014) texture state cache

(2014) <https://patents.google.com/patent/US9811875B2> *Texture state cache*.

Texture state is a set of parameters associated with a texture (location in memory, height/width,

compression/pixel format, style of interpolation/sampling, etc). Since these values are referenced throughout the steps of calculating a texture, and the GPU can switch between textures fairly rapidly, they are held in a dedicated local cache.

Some details we also learn from this are that as of 2015 both address clamping/wraparound and filtering are being handled by the TPU, not by shader code, along with trilinear filtering and at least some version of anisotropic filtering.

The way this seems to work is something like:

- The source code for a GPU shader will include a block of data in memory which describes the Texture State.
- Any particular instruction sent to the TPU references this Texture State in a register
- If the state is present in the texture cache (a tag for one of the entries matches the register) then we note the index of the cache entry, otherwise we evict an entry and load the new Texture State. The entry is then locked in the cache until there are no active instructions using it (tracked by the Pending Counter).
- as the instruction proceeds through the TPU, it references the entry in this cache (a few bits, say five or six, rather than either the 64 bits of an address in memory, or the full 256 bits or so of the entry itself)

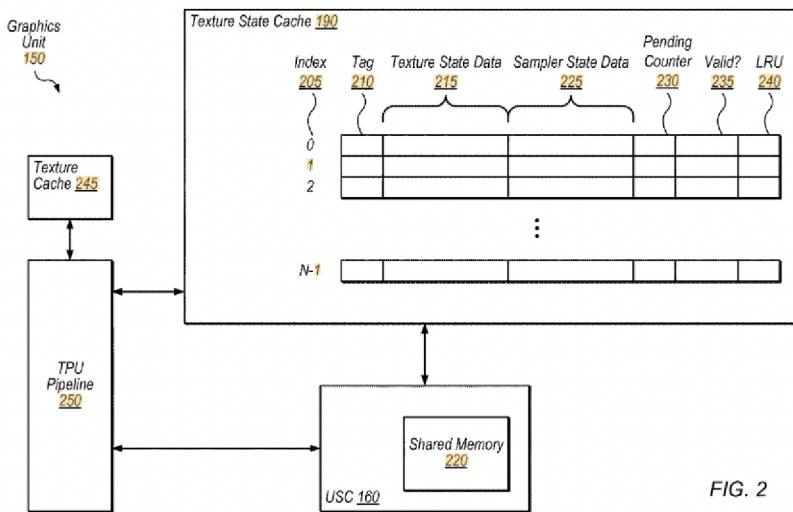


FIG. 2

We also have this neat little diagram which clarifies the TPU stages and how instructions move:

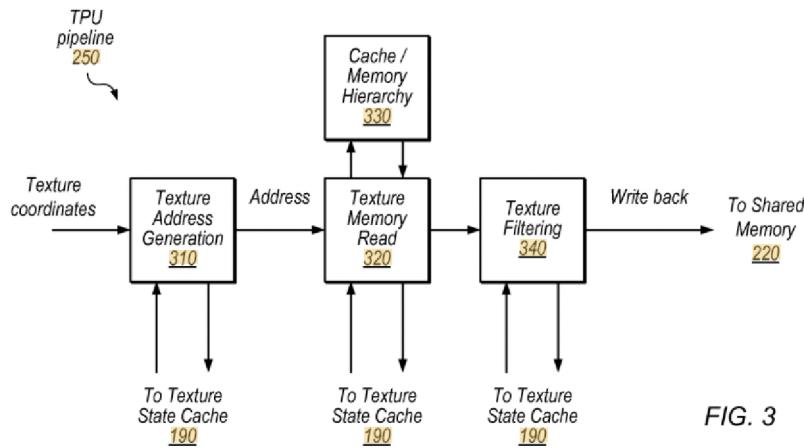


FIG. 3

This sees a conceptual update a few years later as

(2020) bindless textures

Put yourself in the mindset of GPUs 20 years or so ago, when most of the GPU pipeline was hardwired. Consider now how a polygon might be textured; for the texture hardware to extract, filter, and map the appropriate pixels from the source texture to the target polygon various metadata about the texture (width and height, something about MIP levels, something about compression, etc) all need to be known as well as just an address of the texture. This metadata was stored in registers on the GPU (we have seen how Apple uses a Texture State Cache for this purpose), and the process of connecting this metadata from the ultimate resources in the developer's code to these registers was referred to as *binding*.

As time moved on, much of the GPU pipeline became more flexible, but binding continued because while the alternative, of having this metadata stored in some standardized way as a header to a texture and loaded at the start of a shader or whatever, is easy enough in theory, it is a slowdown.

Over the past few years the buzzword *bindless* has appeared in the various graphics APIs, and it refers to, one way or another, removing the limitation of a hardwired number of resources (eg textures) which, for one reason or another, utilize some limited-sized storage pool on the GPU.

The most obvious way to implement bindless is via the graphics APIs, compilers and so on doing the work behind the scenes to ensure that whenever textures are referenced, this metadata is loaded from wherever the standards for the graphics API say it should be stored in memory. But doing this naively is sub-optimal because it's slower than having that data always available in dedicated texture state cache...

So this looks like a caching problem, with a caching-type solution! and that's essentially what (2020) <https://patents.google.com/patent/US20210097643A1> *Bindpoint Emulation* is about – use whatever we can (compiler suggestions, heuristics, counters in the GPU, ...) to decide which textures are accessed most frequently versus which are accessed least frequently, then store the metadata for the common textures in the limited storage space within the GPU, while using the slower loading from memory of texture metadata for the less frequently used textures.

But in a way it's more like use of something like DSID than “normal” caching, in that we don't immediately and continually keep moving texture state between memory and texture state cache, rather we try to maintain the apparently most frequently used texture state in the cache, and the less frequently used state in memory (presumably L1 or L2); and we only move a new texture into the state cache once we have good evidence of frequent reuse.

(2015) address calculation for LoD

Texture mapping begins with the texture itself. If this is the usual case of a mipmapped texture, then we need to figure out co-ordinates into the texture (ie the optimal two level-of-detail maps to use). This is covered by (2015) <https://patents.google.com/patent/US10354431B2> *Level of detail offset determination*.

The main interesting thing this patent suggests is that (as of 2015) the shader was expected to calculate and provide the desired LoD to the TPU, along with the (u, v) co-ordinates. The TPU would, however, handle translating the LoD into an appropriate memory offset for the appropriate (reduced) texture map, along with the appropriate address calculations of (u, v) into the (reduced) texture map.

(2015) texel cache

Then the texture (set of mipmaps) is usually compressed so we need hardware to decompress it. The decompressed blocks of a texture are held in a texel cache, as described in (2015) <https://patents.google.com/patent/US9600909B2> *Processed texel cache*.

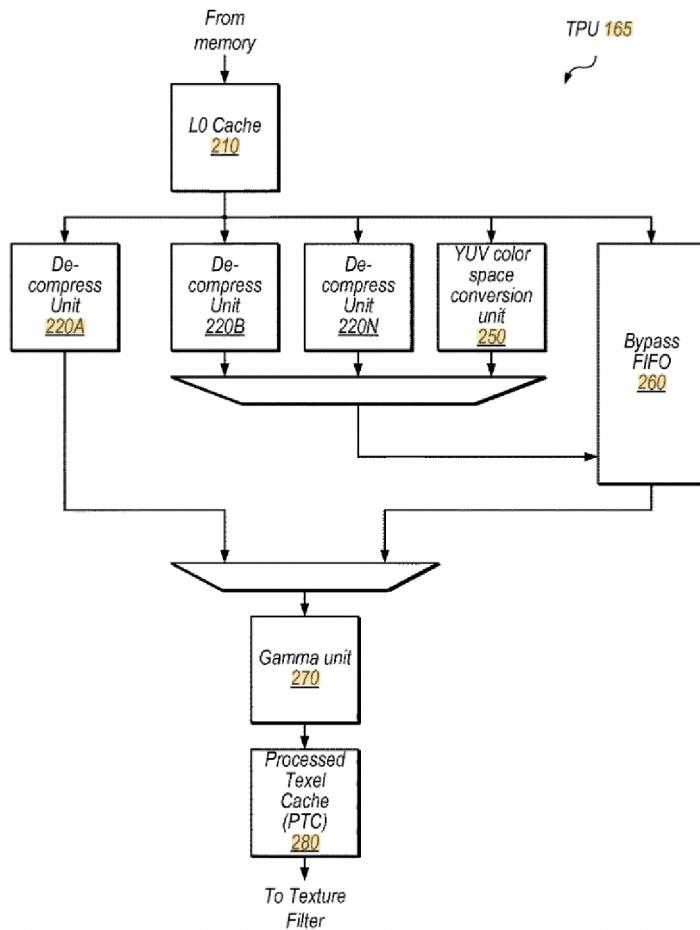
Later we'll see a number of patents related to compression details.

We can see something of the point here:

We have an L0 cache that holds “raw” texture data (compressed, and possibly in a variety of formats), we decompress it and convert it into our ideal processing format, and it is these processed texels that we cache.

The L0 cache is a traditional cache in the sense that it is tagged by memory address.

The texel cache is unconventional in that it is tagged by a combination of textureID+LoD and the (u, v) address within the texture.



(2015) interpolation/filtering (?)

To use the texture involves two steps. First the screen co-ordinate of interest has to be mapped backwards into the texture so that we know the nearest texel values around the screen point. Then some sort of filter ("sampling") has to be applied to convert those four texel values into the pixel value at the desired screen co-ordinate.

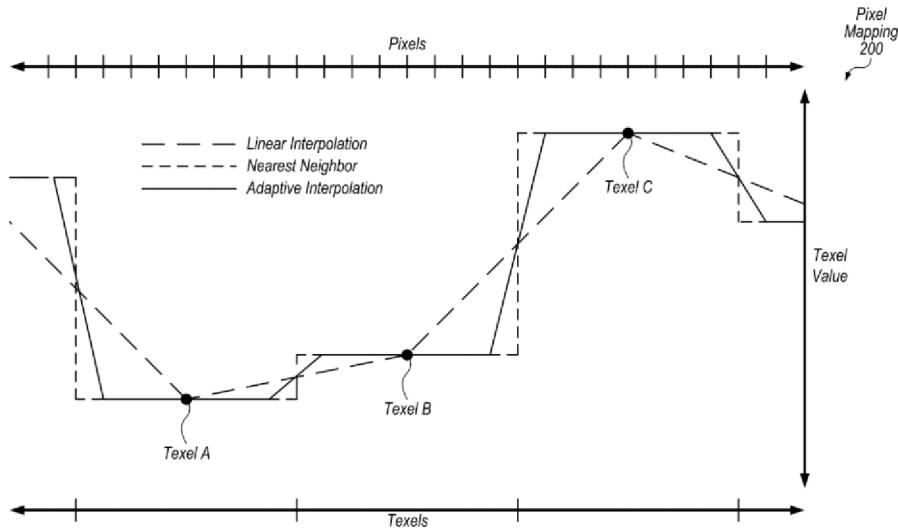
How should we generate an output value based on these nearest texels?

Nearest neighbor is obviously the simplest choice, and just as obviously will result in obvious blocki-

ness.

(bi) Linear interpolation is the next simplest choice, and has long been the standard. But is it visually optimal?

(2015) <https://patents.google.com/patent/US9530237B2> *Interpolation circuitry and techniques for graphics processing* suggests that it's not. Consider the diagram below:



The stair steps of nearest neighbor “interpolation” are obvious, likewise the linear ramps of linear interpolation.

But consider the third “adaptive interpolation” option, which is essentially nearest neighbor near to the texel, and a linear ramp when placed close to halfway between two texels. The patent suggests that most of the time this looks closer to what you really want – the “undiluted” texel near a texel, but a short smooth ramp between texels.

There are then all manner of variants on this idea – how to extend it to 2 or 3D, how to parameterize it (change the width of the flat nearest neighbor part of the curve) and embed that parameter in an alpha channel, so that different parts of the texture interpolate differently, and so on.

This all sounds good, but I can see no mechanism in the APIs for how you access it! Perhaps it's used internally by Apple (eg for rendering text and UI elements) but is considered too specialized for use by general developers?

(2016) region-based sampling

We have a similar uncertain status for (2016) <https://patents.google.com/patent/US10192349B2> *Texture sampling techniques*.

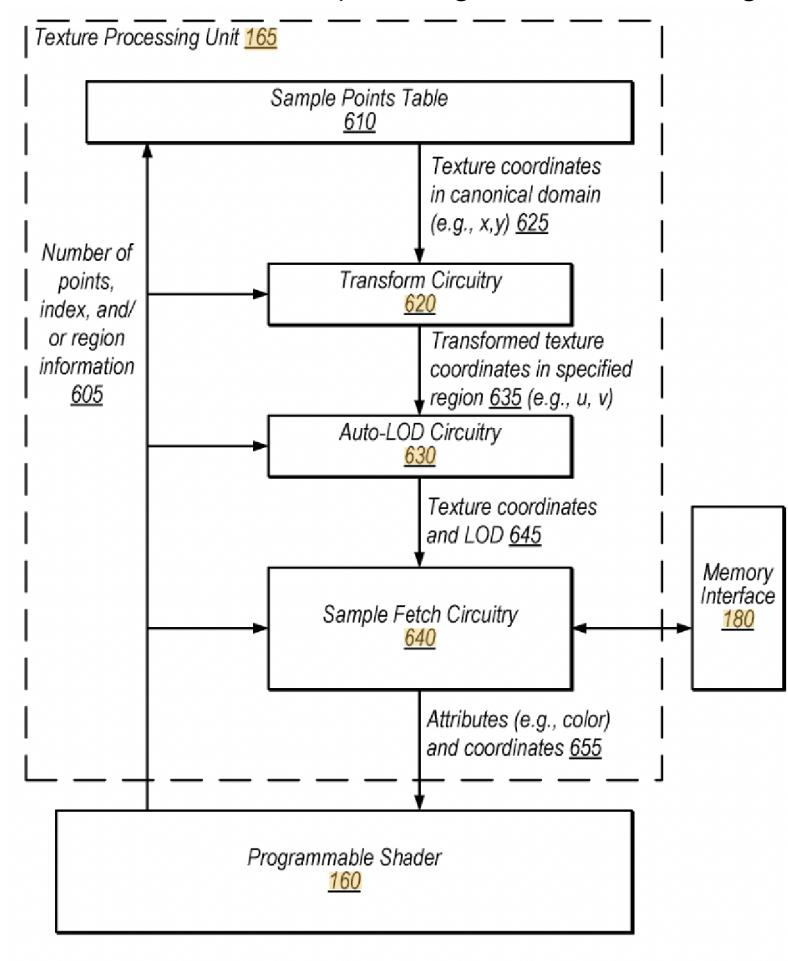
This describes hardware to help with fancy stochastic texture sampling techniques for MSAA.

The big idea is that rather than specifying one (u, v) point after another to the texture unit, meaning the shader has to somehow generate these addresses, you provide the TPU with a region (eg a rectangle in (u, v) space) and the TPU does something appropriate, which could be to super sample (u, v) points within the region using a fixed pattern, or even to generate random points within the region and sample at those points.

Another nice feature of this scheme is that it provides enough detail that the TPU can automatically figure out the appropriate optimal LoD's, something that's not possible when you only provide a single (u, v) co-ordinate.

So all this sounds great, but an obvious constraint is that, in the context of traditional graphics, the regions you would want to describe might be something like the intersection of the triangle with a screen pixel, which is no longer an easy region to describe...

Given the above, I think the way this is designed to be used is that the shader core makes an initial call into the TPU giving a region, and gets back a set of samples along with their locations. It then discards the points of no interest (eg points not within the triangle being textured) and sums the rest to generate the overall color for the pixel being rendered, as in the diagram below:



(The *Sample Points Table* is a table of “optimal” points for sample locations that look pseudo-random and have various desirable properties. You can think of it as essentially a “random sample location generator”.)

There’s another texture patent at this time, (2016) <https://patents.google.com/patent/US20170061570A1> *Graphics processing unit providing thermal control via render quality degradation* which may be obsolete, but is a mildly interesting idea. If the GPU starts to exceed thermal limits, then one option is simply to reduce frame rates, but the patent suggests a more controlled approach by deliberate degradation of the image quality by the texture unit, for example by dropping tri-linear filtering, or dropping MSAA.

(2017) reuse of overlapping texels

(2017) <https://patents.google.com/patent/US10255655B1> *Serial pixel processing with storage for overlapping texel data.*

The GPU mostly processes pixels as 2×2 quads. During the processing of one (u, v) address, four nearest neighbor texels are read (and then filtered to generate one output value for the pixel). This means processing a quad of 2×2 pixels will read sixteen texels.

But many of these read are the same texel! (On average about 9 of the 16 are unique, the other 7 are duplicates.)

The patent takes advantage of these facts to reduce texel reads, rather than blindly performing each of the 2×2 texture lookups independently.

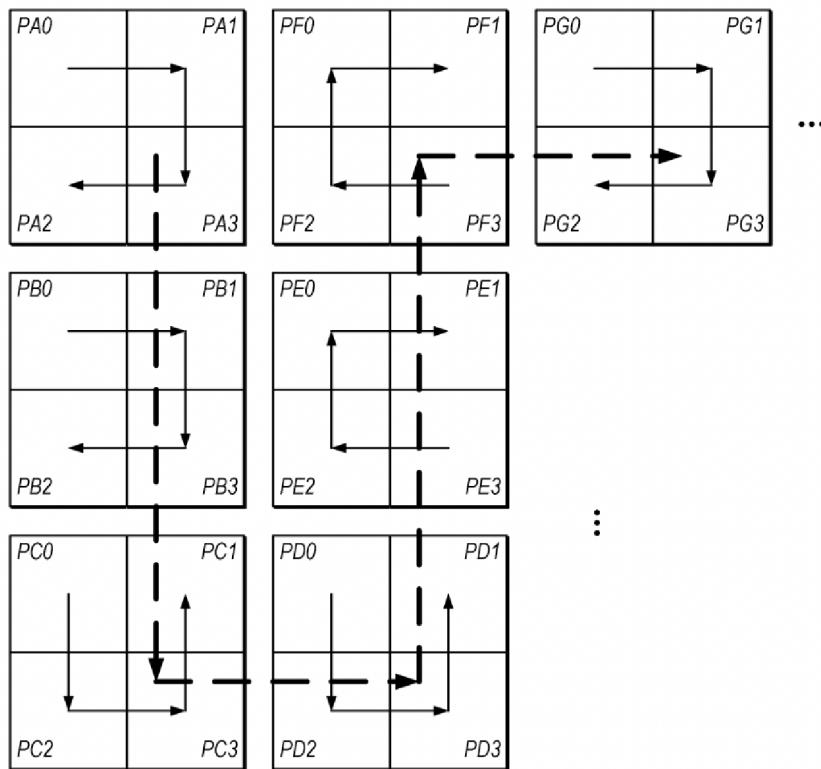
This might sounds like the texel cache, but it’s now storing redundant texels even closer to where they will be reused, in the next cycle or two.

(Don’t get confused. The usual case is that a pixel is fully covered by a triangle. In that case we don’t need MSAA magic, we only need to perform two bilinear lookups for the pixel, one for each LoD, then combine them.)

The more rare case is when a pixel is not fully covered by a triangle. Then we want to activate MSAA and so we will be generating multiple (u, v) addresses per pixel. But the same concepts still hold. Each of those (u, v) addresses will result in four texel lookups, and if we order the (u, v) lookups in optimal order, many of the texel lookups will be redundant and so can be avoided.)

Once you have this very local caching of the last few most recently use texels (say two or four texels), you want to optimize texel reuse. The patent gives an example of how you order sampling for optimal reuse:

Within each quad we process the pixels in the order shown (which varies by quad) in a way that attempts to maximize edge overlap, and we process quads in the order shown, with the same goal.



(2019, 2020) gpu compression of *temporary textures*

As you surely know, GPUs are in a constant fight to reduce memory bandwidth, which can be helped by compression. We've also seen that resource textures should be compressed for file/RAM storage, and are decompressed close to ultimate texture lookup.

But on the reverse side, GPUs are also constantly generating textures, for subsequent render passes, or for composition by the Window Manager. We can save bandwidth if these are compressed on the way *out*, from the GPU to SLC/DRAM.

(2019) <https://patents.google.com/patent/US11062507B2> *Compression techniques for pixel write data* gives the details, but the essential point is that pixel write data is gathered in the L1 cache with, as far as possible, an attempt to aggregate large blocks (4x4, 4x8, 8x8). When data is written back to L2, if a full block is aggregated, then it will be compressed and L2 will hold that compressed block. When data is written back from L2 to SLC/DRAM hopefully non-aggregated blocks will have their missing pieces in

the L2, and possibly smaller (4×4 , 4×8) blocks will have neighbors, and in both cases these will be compressed as much as possible and written out. In the worst case, if data is missing from an L2 block that must be written out, the original block will be fetched from SLC/DRAM, decompressed, the missing data will be merged into the new block, and then compressed.

There are a number of compression schemes used by GPUs, but Apple apparently had reasons for defining their own, described in

(2020) <https://patents.google.com/patent/US11405622B2> *Lossless compression techniques* and

(2020) <https://patents.google.com/patent/US20210336632A1> *Lossy Compression Techniques*.

(Perhaps using the standard texture compression algorithms requires a licence? Or perhaps the Apple scheme is easier to implement for an encoder within the GPU that's trying to encode blocks on the fly as they are written to memory?)

The hardware preferentially tries for lossless compression of each block but when that fails it forces lossy compression.

(2019) separate texture address space and sparse textures

“Traditional” GPUs have a dedicated texture cache, but given our analysis of a generic graphics address space, to be shared by registers, scratchpad, ray tracing, etc, it’s an obvious implication to also put textures in this address space, and to use the common pool of per-core SRAM as cache backing for these textures.

The texture case is interesting just because it’s not one more use case, but because it seems to be the first implementation of this idea of an additional layer of graphics address space. We first see it reference in (2019) <https://patents.google.com/patent/US10872458B1> *Graphics surface addressing*, a year earlier than the generic graphics address space patents.

Now, if that were all there was to it, there would be nothing more to say. The basic ideas seem fairly obvious

- we may want to use large (possibly very large) textures
- we certainly can’t load these all into DRAM simultaneously
- so we want some sort of paging scheme that, as appropriate, pulls in the demanded parts of a texture from some backing store.

Described this way, standard virtual memory (to pull in pages of a texture from a memory mapped file backing the texture) and caching mechanisms (to pull in and discard lines from pages of the texture from DRAM) seem to be able to do the job just fine.

So why (given the specific case of textures) do we need an additional address space, and an additional concept of *Sparse Textures*?

As far as I can tell, there are two issues.

The first one is that the scheme I have described ultimately relies on a page fault being generated when we access a texture that’s not present in DRAM. I don’t think this is an impossible burden. It does require the TLB, sitting between L2 and the rest of the SoC, to be able to detect a page fault (ie access to a page that is legitimately part of the memory mapping for this process, but whose page is not

present in DRAM), pass the fault to the OS, and have the OS page in the data from the texture backing file.

Right now I don't believe the GPU's TLB is capable of this, which has, as one unfortunate consequence right now, that all memory that might be touched by the GPU has to be wired by the graphics driver before a kernel starts, which delays the start of kernel execution and limits how large can be the data objects manipulated by the GPU.

But having the GPU handle faults is not conceptually an impossible task, and it doesn't even require any complications in the GPU below the TLB, to the L2 and the rest of the core it just looks like a data miss to DRAM that's taking longer than normal. So maybe one day...

However there's a second issue, which I think is the real point of this technology, that it's for things like MMORPGs (and perhaps for things that might one day look like them, like annotations to be attached to what Vision Pro sees as you walk through the world...)

The point is that the model I described above assumes that ultimately the full texture of interest is available in a file, and so can be accessed via the "trivial" mechanism of creating a memory-mapping for a texture file and letting the OS handle all the work, including specifically page faults. But what if the backing texture is something like the terrain of an MMORPGs (or some aspect of the real world) that is far too large for files on an individual device, and is streamed in on-demand from a server?

Technically this is still not a deal-breaker. Back in the original days of Mach, one of the design elements was the ability to provide multiple external *pgers*, ie fragments of code that could take a virtual address and provide the data that was supposed to be located at that address by whatever means. Obviously loading that data from a file is one way of providing the data, but another way might be loading that data from a network (or even synthesizing the data, as in eg a random number source, perhaps hooked up to some specialized hardware).

However over the years Apple seems to have deprecated the concept of external pagers, so they did not pursue this particular path. Though, I don't know, like a few other elements of the original Mach design, conceptually it makes sense and you could see it as perhaps one day making a comeback?

So the natural solution to the problem of handling streaming textures would be

- map the texture into some large range of virtual address space
- texture hardware references an address in that virtual address space
- address misses in L1 and L2 (virtually addressed), passes through TLB on the way to (physically addressed) SLC and DRAM
- TLB generates a fault (based on the virtual→physical mapping)
- which routes to the OS
- which routes to an external pager supplied by the MMORPG company

Instead, however, the Apple solution looks like

- define a new address space, which for now we'll call texture address space (later, presumably, this is unified as part of the general graphics address space?)
- texture hardware references an address in that texture address space
- address misses in texture cache (addressed using texture address), passes through texture TLB on the

way to (virtually addressed) L1 and L2

- texture TLB generates the “equivalent” of a fault, namely a detection that we want to access a texture page that hasn’t yet been made available
- ultimately this “fault” is routed to the software supplied by the MMORPG company

So similar in many ways, but differing in the big detail that the “external pager” in some sense lives in and is affiliated with the GPU, not with the OS. A second, more minor and less interesting, difference is in the precise details of how the fact of a “fault” is conveyed to the third party software and how it responds. Part of Apple’s concerns here (as kinda makes sense for the projected application) is

- counters are provided for what access patterns look like for currently present texture pages (so that the third party software can decide which texture page to toss when we load in a new page, since obviously we can’t keep loading next texture pages forever in a finite RAM!)
- the third party software may be able to make good predictions about what texture pages will be required next (think of eg walking in a particular direction in an MMORPG) and can pre-emptively load those pages over the network and inform the system of their presence so that they are mapped into the texture address space.

From the developer’s point of view, this is like the difference between writing code for a virtual memory system versus writing segment-swapping code for a system with limited address space; a whole world of hassle and effort just disappears. Of course the developer has to write the “external pager” to handle “texture faults”, but that’s one piece of code; the rest of the system just deals with an abstraction of textures as large as we like.

(2019) <https://patents.google.com/patent/US20200380734A1> *Graphics system and method for use of sparse textures* discusses all this in a little more detail, suggesting, among other things, that users of this facility, as much as possible, always provide rapid access to a low-resolution MIP-mapped version of the texture, so that this low-resolution can be used to draw for a frame or two while the full resolution texture is being provided (eg over the network).

A generalization of this idea is for an “external pager” to look at the counter numbers before making any decision to load any particular texture page. For example if our players only walked past the edge of a texture tile and have already passed it (or will soon be past it) maybe there’s no point in pulling in a higher resolution version, just accept the low level of detail and move on.

To be fair to Apple, these two elements (the use of MIP-mapped levels of detail in textures, and the fact that textures may be of use for a frame or two, then irrelevant if they were not provided soon enough) are in fact rather different from the contract we expect with traditional virtual memory! We would not be happy with a pager from disk that gave us the first few bits of some data as a “good enough” approximation, or that assumed that it’s taking some time to load this data so let’s just drop the request and move on! You could probably provide enough context to a pager to have it know these details, but maybe it’s better to provide a specialized texture paging service, only in the GPU, but aware of these issues.

Ray Tracing

Ray tracing, like “graphics”, is a large area, which means that accelerating it can be done in many places. You need to understand what the various tasks are to understand what Apple has done, and what’s coming up.

If we want to accelerate a single scene, the job consists of two parts:

- create a data structure (essentially a complex tree), called the ADS (acceleration data structure) or BVH (bounded volume hierarchy), that describes successively tighter bounds of the geometry of the scene. Like any large scene graph, you probably want the ADS to handle instancing (ie multiple copies of the same geometry, but with different locations and orientations).

- fire a large number of rays through the ADS. For each ray test successive nodes in the ADS until it looks like there’s hit in some particular node. In that case, investigate the situation more carefully (test the ray against the exact geometry primitive, not the bounding box) and if there is a hit, do something appropriate (which may be to pick up some color, reflect the ray, bend the ray, generate more rays or whatever).

So if you think about it, ray tracing involves an interleaving between generic work done by the GPU (generating lots of rays and walking the ADS) interleaved with work specific to a developer (something like a callback function for each primitive to handle a ray that intersects the primitive, whether to light the ray, reflect it, bend it, or whatever).

If we wish to accelerate a sequence of frames, then there is an additional third task of handling the frame-to-frame changes. We can usually, for a few frames, slightly modify the existing ADS to cope with the slightly modified geometry of the successor frame. But the bounding boxes of each ADS node become an ever worse fit to the real geometry, and at some point we need to flush the ADS and recreate it.

There is a final optional fourth task of taking the generated frame, which may be somewhat noisy, and denoising it, includes the use of everyone’s favorite buzzword, namely AI; but that’s a somewhat different task that I won’t consider.

With these basic tasks in mind, what Apple has done so far over the years is

- provide API to build (and update) the ADS
- provide API and low level machinery (which over time has moved all the way down to the compiler) to communicate as efficiently as possible what the shader wants to do (for example in terms of looking up material properties of an intersected primitive) and to handle instancing.

So that’s the state of the art as of the M1. There isn’t any ray-tracing-only hardware, but the SW system and the generic shaders are able to achieve adequate performance for many purposes. Most importantly, there is rich API in place for when hardware may become available.

The most expensive task, the first part you'd want to accelerate is intersection testing, and this is what people usually refer to when they talk about "accelerating ray tracing". But once that is in place, there are multiple additional elements you could also improve, including

- building the initial BVH
- modifying the BVH from frame to frame

Another element that could be added is improved random number support. Ray tracing requires an on-going stream of random numbers! The initial ray parameters are random, and each intersection may result in more random rays.

If you want to do animation, you want accurate motion blur which means the time at which rays are launched is an additional random parameter.

Beyond those obvious cases, if you really want to do accurate ray tracing, you don't track a ray as just a geometric entity or even an RGB triplet, instead you generate a few rays with the same geometry but different random frequencies (ie different pure spectrum colors), track how each moves through the scene, and build the final RGB result from the final spectrum at a pixel (approximated as say 16 pure-spectral color points).

So the procedure requires a massive number of random numbers, and making those faster will help ray tracing, but also many other use cases! The simplest support is just the provision of (pseudo-)random independent and uniformly distributed 32b integers. From a bag of 32 random bits, one can construct what one needs; but it costs a few cycles. Ideally why not have the GPU provide support for the most common use cases including

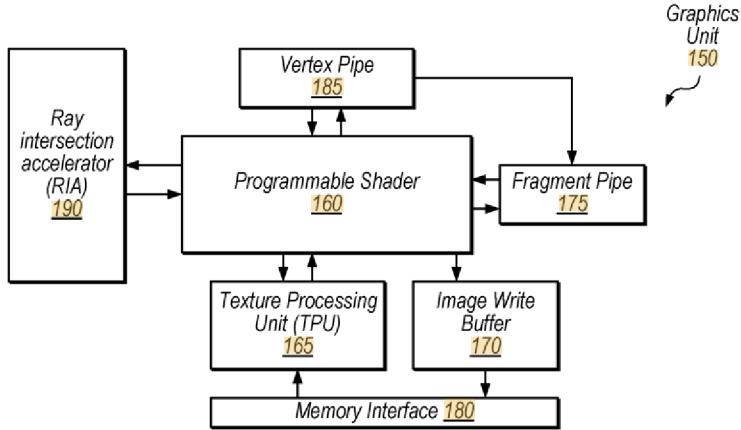
- uniform integers from m to n
- uniform floats from p to q
- normal floats with mean μ and standard deviation σ
- uniformly distributed points on a sphere (equivalent of uniformly distributed directions)
- point sequences from the most common processes, including white noise and blue noise

But enough about what Apple doesn't yet offer. What you all want to know is what Apple does offer, as of the A17/M3, with ray intersection hardware.

(2020) ray intersection hardware

This hardware is described in (2020) <https://patents.google.com/patent/US11367242B2> / *Ray intersect circuitry with parallel ray testing*.

We start by augmenting the baseline GPU core diagram, pretty much unchanged since the very first Apple GPU patents, with a new element, The Ray Intersection Accelerator!



The rest of the patent essentially describes this new RIA.

It's also possible to add hardware to accelerate the building or updating of the ADS, but that is not described. There are a few interesting aspects to the ADS, mainly to make it as small as possible by limiting the number of (fixed point) bits used to describe the bounds of each node. Doing this with a single fixed number of bits might limit the accuracy available to the lower nodes (describing smaller volumes) so there is an implicit shift of the bounds data by some power of 2 that changes as we move further down the tree.

The single most important thing to note is that the RIA is a separate piece of hardware that runs independently of the rest of the core. In a sense we load it up with work to do, let it do its work, then the RIA reports back to the shader core with what it has learned (basically for every ray either “there was no intersection” or “it looks like it intersected primitive N”).

The ADS and the per primitive geometry and material properties live in device memory (ie standard memory, accessible via L1 and L2 caches). However the RIA also has its own private page-based address space, used for various purposes, probably allocated and handled in the same sort of way as described above in the *On-demand Memory Allocation* patent.

What lives in this Ray Address Space is various per-ray data. The most obvious of these data are the ray geometry (ray origin and direction) and a stack used by each ray as it walks the ADS (essentially this walk is a standard depth first tree traversal), but there is also some storage for the RIA to communicate with the shader unit, and some storage to group rays for optimal cache usage.

We'll explain the big picture below, then we'll walk through the most interesting details of the system. If something doesn't fully make sense in the initial big picture rundown, hopefully it will make more sense when we hit the details.

big picture

In big picture terms, what we want to do is perform the common parts of ray traversal (walking the ADS) in the RIA then perform ray shading (ie handle the intersection of each ray with a primitive) in the shader unit. Ideally this will happen in parallel so that both shaders and RIA are constantly busy.

So the idea is that we start by giving the RIA a set of rays generated by the developer (either randomly, randomly but with some sort of importance sampling, or via some pattern that's relevant to the scene) along with the address (in device memory, remember) of the ADS.

The RIA then, for each ray, starts at the top of the ADS and, for each bounding volume node, checks whether the ray intersects the volume. If so we examine deeper in that volume to get a tighter bound, culminating in the tightest bound which gives us the primitive against which we (might) intersect; otherwise we drop this node and move on to a sibling node.

This can be done in parallel, handling each ray independently (each with its own block of storage and ADS traversal stack, living in the Ray Address Space but cached locally) and special hardware can be used to perform all the bounding tests in parallel.

The main thing we need to worry about for this stage is memory traffic, because the ADS is large. So, in fact, each ray is not treated independently from beginning to end.

Rather, in essence, we run over every ray for one step of the process (ie for one node of the ADS) then move onto the next node and repeat for every ray. This is a simplification of the process but gives the rough idea.

Essentially as we process each node, the rays are grouped and placed in group storage (based on which node they intersect), then for the next round the rays are handled by group. The point of this is that we can load in a portion of the ADS that corresponds to some localized region of geometry and test many many rays in that region of geometry against the node before we move onto a different portion of the ADS and test different rays against that; so we can usefully cache regions of the ADS.

The end result of this process is a set of rays (each specified by a rayID) and a final bounding box for each ray (which means one or more primitives in that box).

These are gathered together in the RIA as an array to be passed back to the shader, for example ray 0 needs to be tested against primitives A and B, ray 1 tested against primitives X, Y and Z, and so on.

The most obvious way to do this might be something like allocating one SIMD to each ray, so that each element of the SIMD checks a primitive, up to 32 possible intersection primitives.

But that's suboptimal because usually each ray only needs to test against maybe two to four primitives, so we are wasting most of the SIMD width.

Instead multiple rays (each with multiple primitives) are packed into a SIMD like so.

<u>Ray ID</u>	2	2	2	2	2	1	1	0	0	0
<u>Prim.</u>	9	8	7	6	5	4	3	2	1	0
<u>Count</u>	5	5	5	5	5	2	2	3	3	3
<u>TID</u>	4	3	2	1	0	1	0	2	1	0

Basically the required elements (rayID, primitiveID, etc) are laid out in Ray memory, then loaded into Shader registers, then each shader lane can execute some code to test against the primitive geometry, find the appropriate hit (the ray may hit more than one primitive, so we need to earn the nearest), then load the hit primitive's material properties and "shade" as appropriate.

(The parts that "matter" are the Ray ID and Primitive. The other fields are for "management".

Count says that a particular ray has 5, or 2, or 3 primitives associated with it.

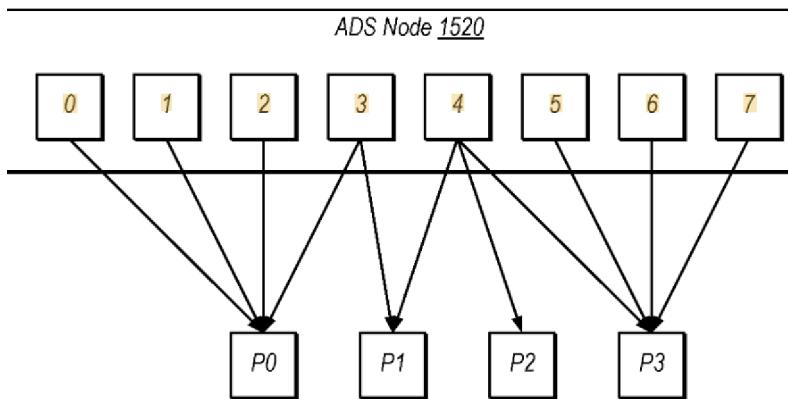
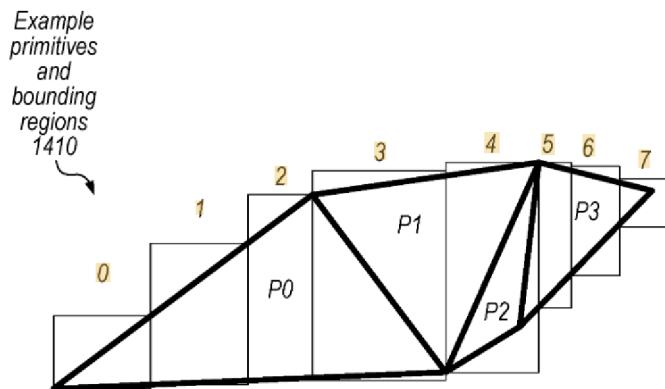
TID indicates the "test number" for each ray, whether that lane is testing, eg, the first, second, or third of three primitives.)

One interesting thing that makes this work is that the multi-lane Reduction operations (eg max or min) can now be scoped to "segments" of the SIMD, so that eg the max operation across a set of lanes will operate independently within the lanes of rayID 0, rayID 1, and rayID 2! The patent does not indicate whether this functionality is available outside ray tracing, but does hint at how the segments are described, so it seems like this advanced "Scoped Reduction" functionality could be generically available in A17/M3 GPUs.

Another interesting detail is that if geometries are instanced, they do not have to be "flattened", ie converted to world geometry, for final primitive testing, instead the ray is transformed into the instance geometry (and then, if relevant, transformed back to world geometry on exiting the shader). The patent says this is overall cheaper than transforming instance geometry to world geometry.

Another thing that the patent talks about a bit (so maybe it is unusual) is the precise details of how the ADS is constructed, with a single node (ie bounding volume) able to cover multiple primitives, but *also* to only cover part of a primitive (meaning that a single primitive could "sit in", ie be referenced by, multiple nodes). The claim is that this allows for much tighter geometry bounds so that almost every ray that leaves the RIA for ray shading will actually be a shaded ray, it won't be discarded in the shader as not intersecting any primitive.

So in the diagram below, a ray that intersects any of boxes 0, 1, 2, or 3 will be routed to a hit test against primitive P0, while an intersection with box 3 will *also* be routed to a hit test against primitive P1, ie an intersection with box 3 will result in two primitive tests not one. Likewise an intersection with box 4 will result in three primitive tests.



details

So given what we have said, to make ray tracing as efficient as possible

- we want to fill up the RIA with as many rays as possible (so that each time we pull in some geometry of

the BVH) that geometry gets tested against as many rays as possible

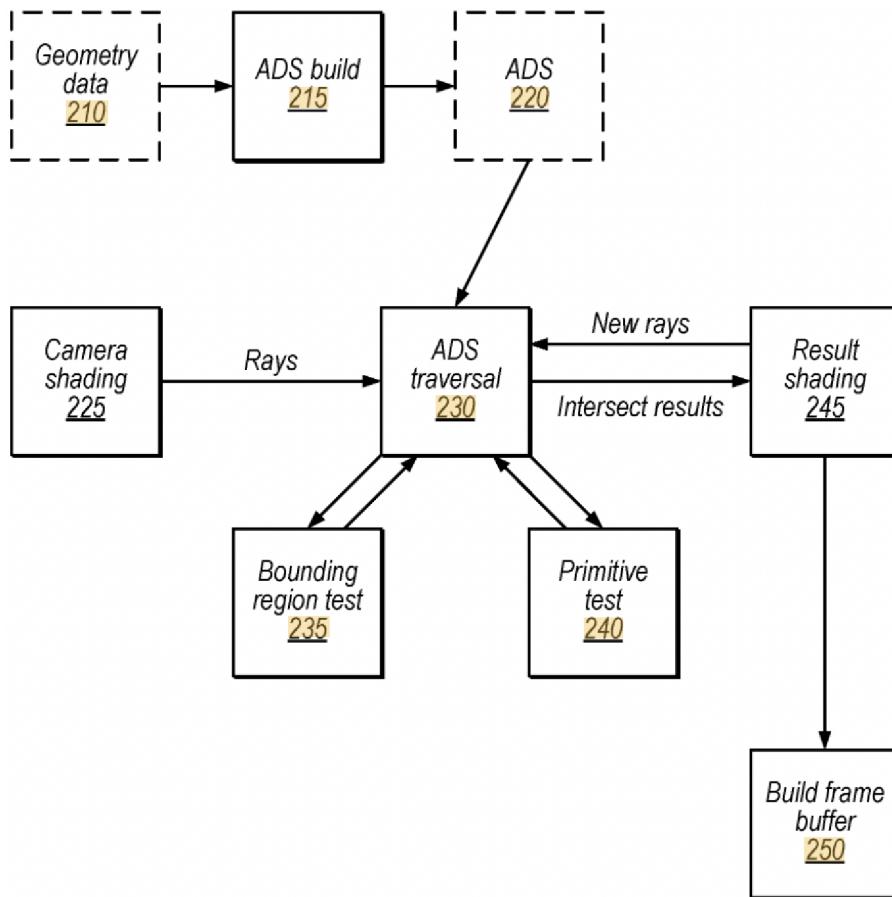
- we want to keep rearranging rays within the RIA so that we have clustered together in each parallel bounds test a whole lot rays in the same region of geometry

- we want the bounds returned to the shader core to be as tight as possible (ie if we give a list of primitives that the ray might intersect, because those primitives all sit in the same leaf node of the BVH) we want that list to be as short as possible

Tools for the above include

- various caches in the RIA
- clever (and perhaps original to Apple?) ways of using the BVH

We start with the data flow:



The above recaps what we have said.

- We convert the geometry in an ADS.
- The shader core (following code written by the developer) generates a set of rays and passes them to the RIA
- The RIA walks through the ADS, testing groups of rays against nodes, and eventually arrives at a list of which nodes (apparently/maybe) intersect which primitives
- This list goes back to the shader core which responds based on the material properties of the intersected primitive. (Maybe the ray is absorbed, ie generate a color. Maybe it is specular reflected. Maybe it is refracted. Maybe it is scattered, resulting in the generation of multiple new scattering rays.) In the

cases where new rays are generated (scattering, reflection, refraction) those rays go back into the RIA to be traced again. This continues until a ray is absorbed or some proxy for absorption kicks in (eg we assume that after say four or six generations the ray is so dim we no longer care what happens to it).

The scene used to construct an ADS is usually not flat, instead it is formed hierarchically, eg imagine an island that has a bunch of trees on it, each tree has a bunch of leaves on it, and so on. Even though the island has say 30 trees, there may in fact be only three basic tree models, each one replicated in the scene with slightly different geometry – different location, different angular orientation, different scaling along one, two, or three axes. Likewise each tree can be made out of a few basic objects repeated with varying geometry.

You can build the BVH by flattening each object down to its primitives. But Apple claim it is more efficient to retain the hierarchy.

So instead of having a ray retain its geometry through every node test, against a constant world geometry that's the same in every node, rather occasionally the ray geometry is “rotated” to match the model geometry. in other words instead of the ray seeing a tree that has been turned sideways, we turn the ray sideways as soon as we cross the node boundary into this particular object and its particular geometry. Doing this allows the ADS to be a lot smaller because it can reference one model (with variant geometries in different nodes) rather than many many triangles resulting from the flattening of each model.

This instancing *may* also make it cheaper to modify the ADS from one frame to the next as long as motions are small, by simply updating the geometric parameters of each instance of each object, rather than updating the geometry of every primitive in every object.

Note what all the above means in terms of data flow. The RIA proper is basically blocks 230 and 235 above.

Every so often the RIA will steal a few cycles from the shader core to transform the geometry of a set of rays when those rays cross an instancing boundary and need to be transformed from world geometry to model geometry of the model within that ADS node.

And once an apparent intersection is arrived at for a set of rays, they will be sent back to the shader core for primitive hit testing (step 240).

The patent is (as usual) devoid of numbers, but the idea seems to be something like

- a shader core “allocates” a set of rays which essentially allocates storage for each ray in ray address space, and returns a register of rayIDs
- based on the rayIDs (which route to locations in ray address space) the shader calculates the geometry desired for each ray and stores it (as usual, calculating and storing 32 lanes wide) in ray address space
- a shader core executes an“intersect ray” instructions passing in the register of rayIDs.
- By making repeated such calls we fill up the RIA with some number (128? 512?) of rays.
- the RIA now repeatedly and autonomously
- + sorts the rays by geometry (gather “nearby” rays together)

- + tests some number (8? 32?) of rays simultaneously against a node of the tree
- + walks to the next node of the tree depending on whether the previous test indicated intersection (or not) of the ray with that particular subvolume of space

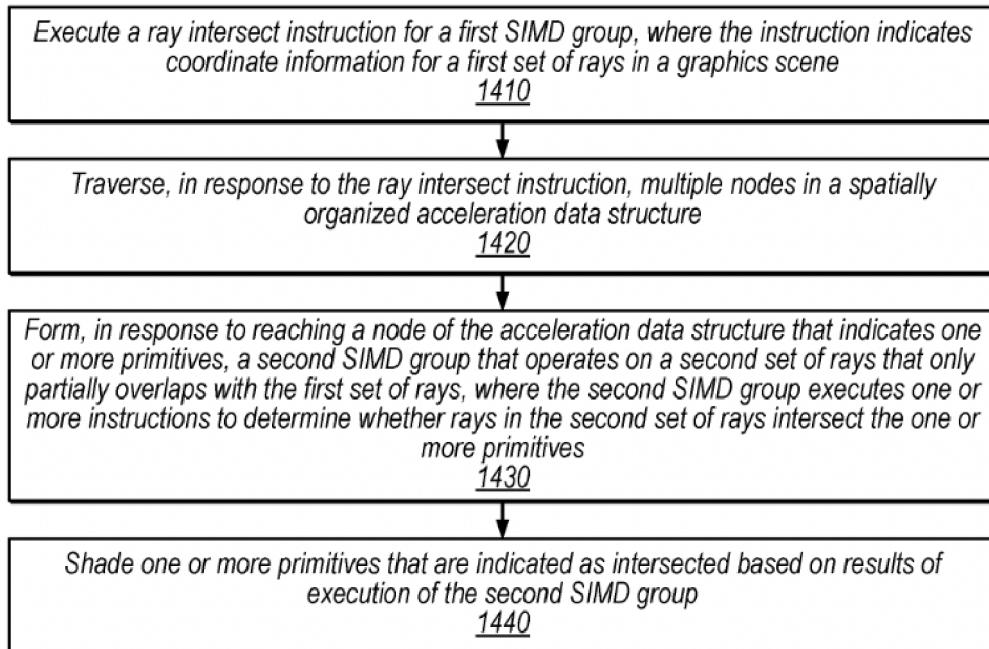
You can read a ray tracing book for the details; the specific point of interest here is that you can start by imagining in your head how you would walk such a tree (think of eg an octree) for a single ray, then imagine running that operation in parallel. If the rays all originate, and point in different regions of space (are “incoherent”) then you will have something like SIMD branch divergence in the shader core. But if we repeatedly sort the rays to try to ensure that they are “coherent” we can mostly run a full set of tests against a full set of rays (8? 32?) with every test active most cycles, and rarely having to mask out some test lanes as inactive.

The bounding tests are done in lower precision arithmetic (with custom circuits that round upwards or downwards depending on which side of the box is being tested) so that false positives may be generated, but we’ll never miss an intersection in a particular volume.

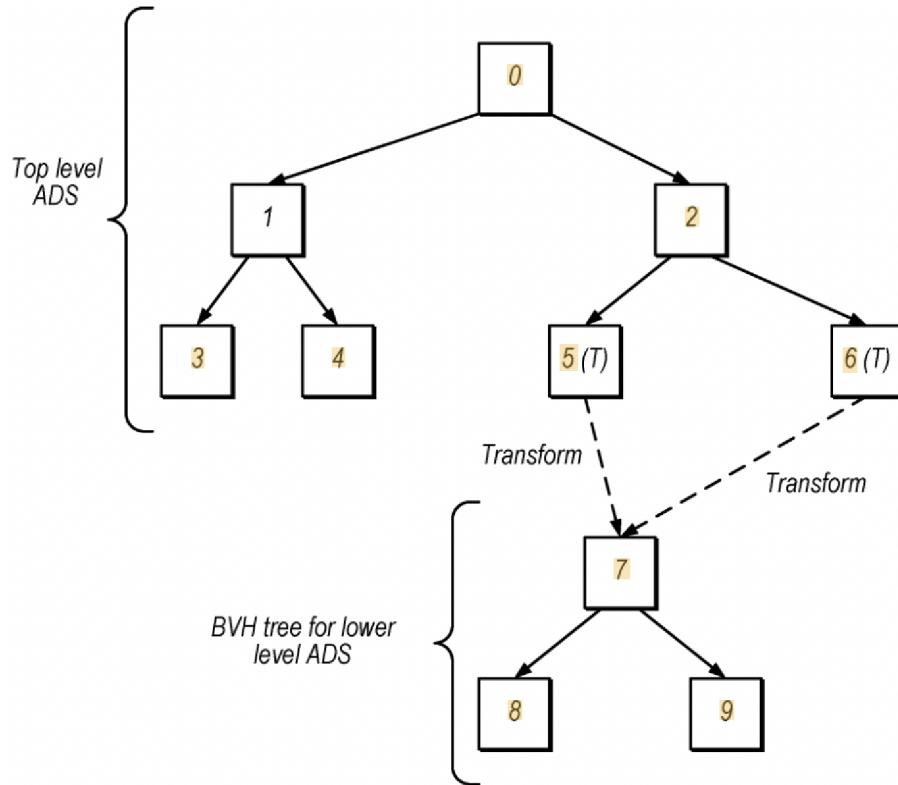
On reaching the leaf nodes of the ADS, we can see what primitives are associated with that leaf node and, as described (recall

<u>Ray ID</u>	2	2	2	2	2	1	1	0	0	0
Prim.	9	8	7	6	5	4	3	2	1	0
<u>Count</u>	5	5	5	5	5	2	2	3	3	3
<u>TID</u>	4	3	2	1	0	1	0	2	1	0

the shader core can ultimately generate for each ray the intersected primitive (or no intersection) and can then make a decision about what to do for each ray (reflect, refract, etc). So basically what’s described below:



The diagram below perhaps explains the issue of instancing better.



Suppose our scene has, say, two cars in it. A car model is represented by the second-level tree composed of nodes 7, 8, and 9.

So a ray walks through the tree seeing if it intersects the volume of space represented by node 1.

No, so does it intersect the volume of space represented by node 2? Yes. So maybe it hits one of the cars. Which one?

Does the ray intersect volume 5? If so, transform the geometry of the ray from world space to model space, and now walk the 789 tree (representing a car) to see which region of the car it intersects. Maybe no intersection. But the ray may intersect car two. So check volume 6. If there's also an intersection, transform the ray geometry again, from world space to model space. This will be a different transformation because the placement/orientation of the second car in world space is different. Then once again walk the 789 tree, using this second ray geometry. Once we achieve a hit, as is usual with instancing, we will know both the model we hit (which primitive of the car) and the instance number (so we can look up details specific to that instance; perhaps the first car was painted with texture A, and the second car

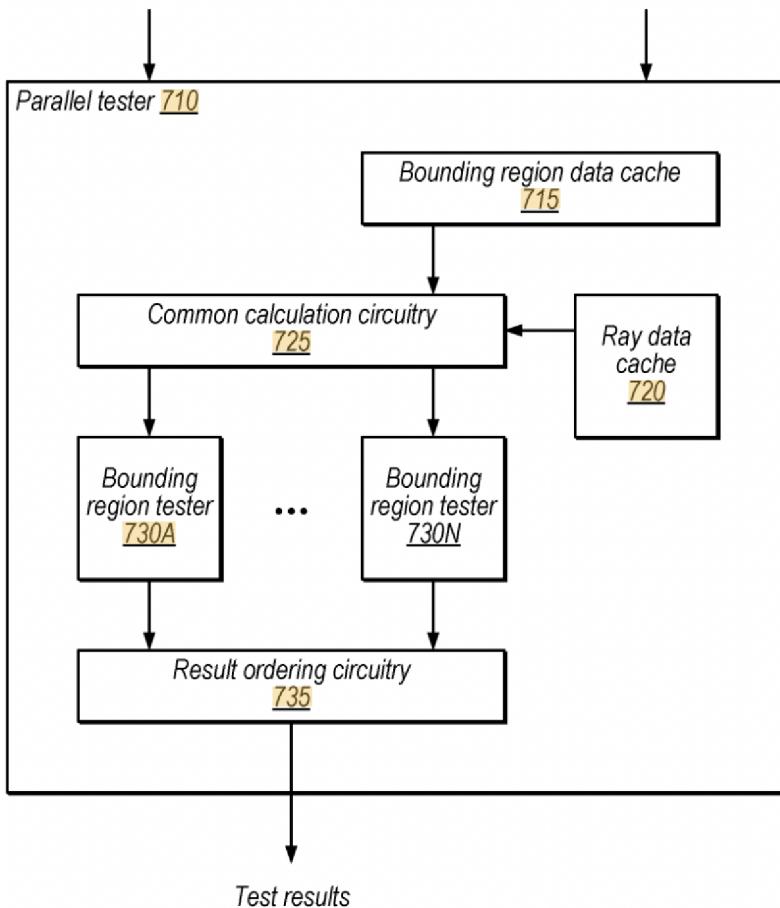
with texture B, so we will look up texture B for the color the this intersecting ray).

The net result of all this is that we probably perform more calculations (each ray that crosses from world to model space is transformed, instead of a one time transform of every vertex in the model, and there will probably be more rays than vertices) but calculations are cheaper than data movement, and doing things this way we land up with an ADS tree that occupies a *lot* less memory (though certain subtrees of it like the 789 tree are traversed multiple times). We save memory traffic, but don't save in number of node traversals.

This also now explains something of the high level structure of the RIA as in this diagram:

Bounding region data

Ray data



If we were walking a traditional tree it's not clear quite what value is there is in a tree node cache as in

715, because traditionally you would walk a region of the tree then discard it. Maybe a small cache to hold nodes close to the root that you might return to after exploring their children?

But with instancing, this cache becomes a lot more powerful, because you will land up re-walking subtrees like the 789 tree multiple times from different nodes, and if you can capture these model subtrees in a cache, you'll do very well in terms of delays waiting on memory, and energy.

There is one thing that bothers me about this scheme that I haven't seen explained.

Obviously every time we cross sub-trees the set of rays needs to be transformed, and the patent says that's done in the shader core.

But I see no explanation of how this is achieved.

Obviously the least satisfactory solution would be that the shader core just sits unused, waiting to be utilized to occasionally transform rays!

The alternative is presumably something like the Metal Compiler creates an "auxiliary" threadblock or something similar, some block of additional code, that sits queued up in the shader core, like any other kernel waiting on memory, then every so often a flag flips, this block of code becomes executable, and it executes its little loop (look at a particular region of shared Ray Address Space, load the geometry of the rays stored in that region, transform them as directed, store them back, flip something that indicates you have finished the job)?

But details of this are not given.

Next we get details on how ray coherency is improved. The basic idea is that

rather than

- choosing a set of say 32 rays, comparing them all against node 1, seeing that half of them intersect and half don't, then using masking to further test the 16 that do intersect 1 against nodes 3 and 4,

instead

- we run a set of intersection tests against all the "rays active for this node" say 512 of them. After this set of tests, we organize the rays into the set that intersects node 1 and the set that doesn't. From the 1-intersectors, we then test (8, or 32, or whatever, at a time) against node 3, and so on and so on.

After every intersection test we re-arrange rays so that the full width of simultaneous bounding-box tests occurs against a full set of rays; as opposed to the situation we have with traditional GPU SIMD where, after a few rounds of if-then tests we can have masking applying to only two or three active lanes and every other lane is inactive.

We can do this because inside the RIA we deal with *rays* not with *lanes*, we can rearrange the rays as we wish, and all we need to throw out at the end is an array of rayID's vs primitiveID's to test against. Likewise when we perform geometry transformations partway through the ADS tree-walking process, all the shader needs to do is modify geometry (values in each lane) as required, there's no requirements (and no such data is provided) to link the geometry being transformed to a particular input ray, let alone to a particular input lane.

You should also remember that the RIA has multiple independent actors executing simultaneously. So we have, for example,

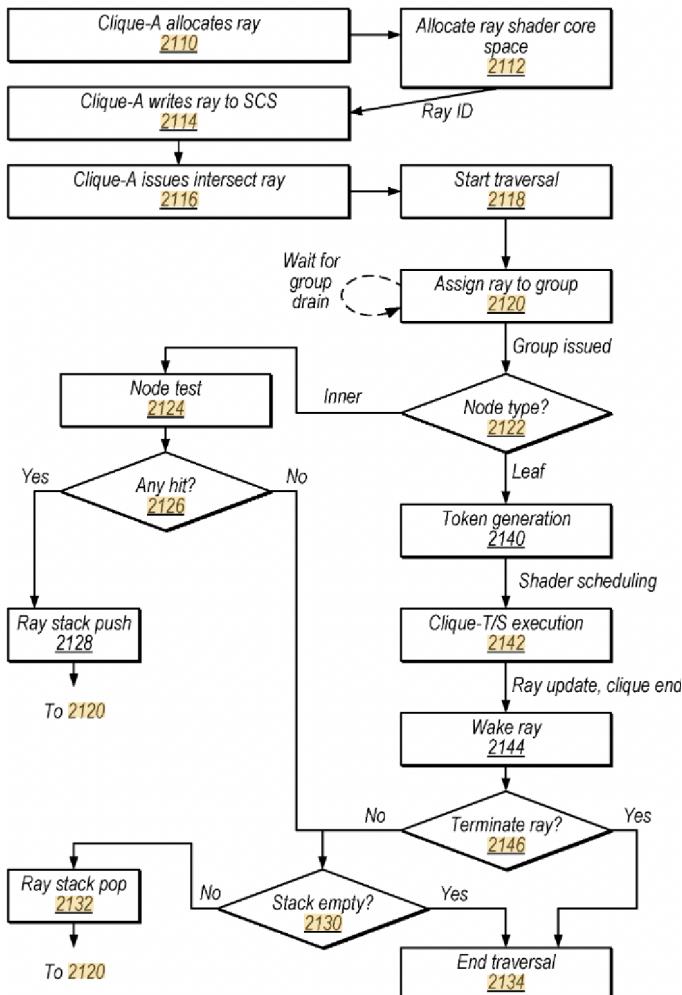
one agent that's loading future elements of the ADS tree and waiting for those loads to arrive.

Another agent is testing groups of rays against the parts of the ADS tree that are cached on chip.

A third agent is walking through the pool of rays looking at the results of their most recent intersection tests, and collecting them into new groups for the next round of intersection tests.

The same sort of sorting happens at the end, when rays are returned to the shader core for final shading; an attempt is made to group together rays that hit in the same final leaf node (ie appear to intersect the same region of space/same primitives) in the hope that the shader that ultimately shades these will mostly execute branches in the same way (ie each ray will hit same result when testing each lane's "is this glass? is it metal? does it refract, or reflect, or scatter", etc).

You can see all this laid out below, and by now you should be able to follow the flow. I've not bothered with the terms Clique-A,-S-T, but these refer to different warps executing on the shader core. A Clique-A submits rays into the system, a Clique-S is a warp that performs geometry transformations, and a Clique-T is a warp that performs precise hit-tests given a list of rays and per-ray primitives that the ray might intersect. Tokens and SCS refer to the details of how the RIA synchronizes use of the Ray Address Space, with and transfers information, to the shader core.



Now how exactly is this laid out? At an abstract level, you could imagine this RIA (Ray Intersection Accelerator) as an IP block associated with anything from one quadrant of a GPU core to an entire GPU core to a few (maybe two or four) GPU cores to one RIA for the entire GPU. nVidia associate their RT accelerators (basically the same thing as Apple's RIA) with a GPU core, and Apple seems to be doing the same thing (though I could see it happening depending on how different work balances out, that we associate one RIA with two side-by-side GPU cores, so that the now doubled-in-size RIA has larger pools of work to operate on while it is waiting for memory, , and can create larger coherent groups all the way

down to the leaves of the ADS...)

refinements

So that's ray tracing hardware 101 – probably rather more complex than you expected! But that's simply the beginning (though my guess is this what we probably have in the A17/M3). There are multiple refinements possible, some obvious, some less so.

motion blur

One element of Ray Tracing that we've so far omitted is the issue of time.

This is not just about generating a sequence of frames. You can do that, but the results looks unnaturally crisp because they does not exhibit any motion blur.

We can fix this by a combination of

- attaching a time stamp to every ray and generating rays at random times within the frame interval (so that the set of rays accumulated over a frame correspond to different times, not a single "frame start" time); and
- in some way incorporating the motion of the geometry in the scene, so that the ray for time T sees the geometry (at least approximately) as of time T , again not just at the frame start time.

The API for this was added to Metal fairly recently, so that part is already in place.

There's also a patent for handling this, 2021, <https://patents.google.com/patent/US20220301254A1>

Temporal Split Techniques for Motion Blur and Ray Intersection. The work is split between a slightly modified version of the ADS (so that each node cover the full volume of each primitive as it moves over the duration of a frame), and work by the shader (to test the ray intersection against the exact placement of the primitive at time T). However the details go very much into ray tracing algorithm specifics of exactly how you can optimize this idea, and are no longer generalizations about the GPU, so we'll not discuss it further. To make this idea work efficiently does require one change to the RIA which we'll discuss below.

This gets updated in (2023) <https://patents.google.com/patent/US20250191275A1> *Hardware Acceleration for Motion Blur with Ray Tracing*.

The previous scheme used what we might call a union volume to cover each primitive as a *union of all* the volumes the primitive might move through during the time interval of interest (ie the frame, 1/60th of a second or whatever) over which rays are being cast.

What the new scheme essentially does is

- record the vertices of the (non-union) bounding volume at time t_0 and time t_1
- position the time of the ray, t_* , as some fraction between t_0 and t_1
- linearly interpolate the bounding volume to time t_* based on the offset fraction of t_* and the vertices at times t_0 and time t_1

Obviously this results in a smaller test bounding volume for each ray (good!) at the cost of requiring the additional linear interpolation hardware.

precision refinements

The design above ran all bounding volume tests within the dedicated Ray Intersection unit, but delegated the *final* primitive intersection tests to the Ray Shader code, running on the compute shader lanes. This split was required, in a sense, because the Ray Intersection unit is doing its work in lower precision (both for the ray and for the bounding volumes), with careful use of rounding modes (ie whether various comparisons are rounded up or down) to be sure that the “safe” choice always occurs in each test (ie a ray that just grazes a bounding box will always be considered to intersect the box, subject to a safe later exact test on the shader).

(2020) <https://patents.google.com/patent/US20220207690A1> *Primitive Testing for Ray Intersection at Multiple Precisions* describes (in complicated full detail!) ways to transform each primitive into that same reduced resolution representation, and so use that same machinery to perform the final intersection test within the Ray Intersection unit.

If this “generous” test of ray against the primitive shows no intersection, then we don’t have to waste energy and time transferring the ray to the shader to have it tested there for exact intersection.

So (to throw out *totally* made up numbers!) the first Apple design generates say a stream of rays+primitives where say 90% of the rays finally tested by shader core intersect a primitive and go on to Ray Shading, and this new design allows that number to rise to something like 99% of rays tested by the shader core intersect a primitive and go on to Ray Shading, with the other 9% of previous rays discarded within the RIA and not wasting shader core cycles.

(2021) <https://patents.google.com/patent/US20230099114A1> *Quantized Ray Intersection Testing with Definitive Hit Detection* is an even more sophisticated version of this idea.

Now, instead of using careful rounding to ensure that the ray is tested against a “generous” version of the primitive (that is, one that is “rounded outwards” by the quantization, so that it appears to be slightly larger than the real primitive in every direction), the primitive testing is done using *interval arithmetic*, which can sometimes give a definitive result as to a hit.

In other words, the hit test can now generate three outcomes:

- + definitely no primitive intersection
- + possibly a primitive intersection (these two were the outcomes of the previous patent)
- + definitely a primitive intersection

This can be used in at least two ways.

One is that the fact of intersection can be attached to the ray, or to the ray shader per-lane register setup, in some way so that this test does not need to be redone.

The second is that this same machinery of interval arithmetic can be used to speed up the testing of time dependent rays against *moving* primitives, ie the motion blur problem that we described a few paragraphs above.

This patent also describes two minor tweaks to the pre-existing design.

The first is that the ADS, which now stores primitives (in quantized precision) for the RIA primitive hit test, stores the primitives, when possible, as triangle pairs rather than as triangles. A triangle pair

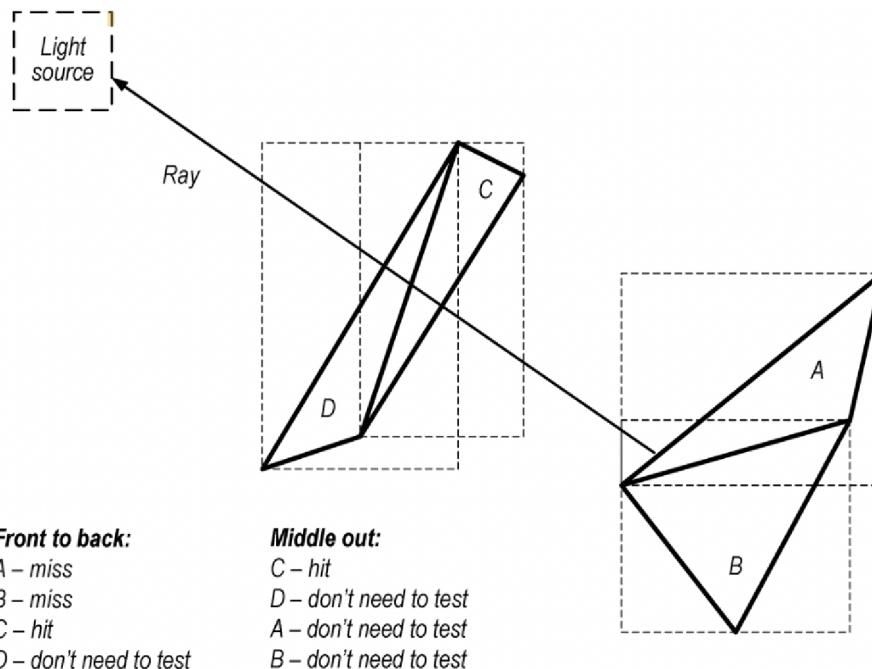
sharing an edge requires four vertices of storage, as opposed to six vertices of storage for two distinct triangles. Obviously this is still not optimal; optimal is a list of vertices, and a separate list of triangles, where each triangle is defined by three indices into the vertex list. But, one step at time...

The second tweak has to do with the order in which we walk the ADS.

The natural thing to do, given a ray, is, conceptually, to start at the eye and walk outward (ie forward) along the ray, and then to translate that into the geometry of how we order traversal of the ADS.

But “on average” we would expect most intersections to occur in “the middle” of the spatial volume, not near the eye. So if we start traversal in “the middle” of the ray, rather than at the beginning, we might expect to do more hitting of relevant bounding nodes, and less skipping of irrelevant nodes. The diagram below shows the idea.

The patent describes how we can use this insight to create a somewhat more efficiently ordered ADS traversal.



Let's make three final points before we move on.

- People tend to think of ray tracing as creating whole scenes from scratch. That's certainly possible, but what we will probably see at first is the use of ray tracing to polish up scenes that are mostly drawn using traditional GPU techniques. For example traditional GPU shadows are faked using various unsatisfactory techniques, but it's fairly simple and fast to add a ray tracing pass to a scene to gather up data for where to draw shadows. You can do something similar for say, one round of reflections, again to make an already satisfactory image look that much better without costing too much extra time.

- It's always interesting to compare with the competition. Imagination have defined a sequence of

levels for ever more sophisticated ray tracing hardware, and this is how they see things as of late 2023:

PowerVR Photon Ray Acceleration Cluster

Ray Tracing Efficiency Summary

	Level 2	Level 3	Level 3.5	Level 4
Example implementations	AMD GPUs/ Console GPUs	NVIDIA <RTX40-Series	Nvidia RTX40-Series Intel ARC	PowerVR GPUs
ALU Offloading	Partial	Full	Full	Full
Hardware Box Testers	Y	Y	Y	Y
Hardware Triangle Testers	Y	Y	Y	Y
Hardware BVH Processing	N	Y	Y	Y
Ray Coherency Sort	N	N	N	Y
Thread Coherency Sort	N	N	Y	Y
Cache Hit Rate	Low	Low/Medium	Low/Medium	High
Memory Latency Tolerance	Low	Low	Low	High
Processing Efficiency ALU	Low (SIMT utilisation)	Low (SIMT utilisation)	High (Thread Sorting)	Very High (Coherency Gathering)
Mobile Power Budget	No	No	No	Yes

In spite of what the internet was absolutely sure was the case after Apple stopped shipping PowerVR GPUs, Imagination and Apple seem to continue to be very close, and if you read up on IMG's ray tracing design, it seems very similar to what Apple is doing; most importantly the part that IMG talk about the most as their special technology, what they call Ray Coherency Sort, ie re-grouping rays after each box-test, is also present in Apple's design.

Where Apple seem to have added their own functionality to this design is using low precision arithmetic for the testing, which initially seems to have prevented triangle testing within the RIA, then allowed it with conservatively-quantized triangles, then allowed even better support with interval arithmetic.

- Finally ray-tracing is not the be-all and end-all of graphics. It solves many problems, but it's not an especially efficient solution for "global illumination" is handling light that comes from large extended sources (think of eg a glowing furnace) or light that results from general scattering around a scene off many many surfaces. There is a better solution for this called Radiosity: [https://en.wikipedia.org/wiki/Radiosity_\(computer_graphics\)](https://en.wikipedia.org/wiki/Radiosity_(computer_graphics))

Radiosity is not mentioned much in "popular" graphics, eg in the context of games or consumer GPUs. Perhaps we will see it mentioned more once people have had time to absorb the consequences of being able to add ray tracing to their existing graphics? Selfishly, I'd like to see radiosity pushed more as a hardware feature because most of the hardware required to accelerate radiosity is probably useful to

generic linear algebra and PDE calculations, so it has value to physics and engineering uses of GPUs.

Other Graphics Details

(2012) apple's TBDR implementation

I've tried to avoid anything too graphics-specific (not least because I know zero about the field) but if this sort of thing interests you, we start with (2012) <https://patents.google.com/patent/US20130293544A1> *Tiled Forward Shading with Improved Depth Filtering*, which is basically "here's how we do TBDR".

The patent starts with the basic point: as we generate geometry and construct primitives in world space, we bin them by x, y into tiles, then draw the primitives in each tile, using a z-buffer to ensure that only the frontmost pixel (let's ignore transparency for now) is drawn.

But there are many details beyond this, for example how do you handle lighting? There are many alternatives, but the particular Apple scheme runs a first pass within a tile calculating various details relevant to lighting (like triangle normals) and accumulating these in a *G-buffer* stored in the tile (ie in fast local memory) then running over the set of lights and applying each light to each primitive to figure out the color of each pixel.

Even with this rough plan in mind, there are multiple ways to make the scheme more efficient, some of which are described in the patent. Ideally you'd like to sort the primitives in some way so that you know which primitives will be overwritten, and so don't need to be drawn at all, but doing this perfectly is close to as expensive as just drawing everything and letting the z-buffer handle all the work. But there are "imperfect" ways to do the job that will save a lot of work. One of these is to bin the primitives associated with a tile in various ways. For example we can bin them into groups, based on how they were submitted, taking into account that most triangles are collected together in space to form a single object (a player, a car, whatever), and we can often make useful decisions at this *object* level; eg we can now associate a bounding box with a set of primitives, and if one bounding box is completely occluded by another, we can drop the second object.

Another way this works is that most lights in a scene can be associated with "active" volumes, eg a sphere or a cone within which the light is active, and beyond which it's so faint that we ignore the effects. We can then see which objects (ie higher level bounding boxes) intersect with which lighting volumes, and only bother with calculations for those objects (and hence primitives) within a lighting volume (ie objects that will be visible).

(2017) fragment generation

Getting more sophisticated (2017) <https://patents.google.com/patent/US10269091B1> *Re-using graphics vertex identifiers for primitive blocks across states* shows some of the details of the vertex processing system. To my eyes, the details look mostly what you would expect.

Most significantly, there is an emphasis on trying to store and reuse vertex processing rather than the more naive approach of handling every vertex within its triangle, which will lead to most vertices (shared by triangles) being handled twice. So you can think of it as something like a fancy cache holding vertices that are expected to be reused soon while generating fragments from primitives.

In a way this matches the *Serial pixel processing with storage for overlapping texel data* patent, which likewise tries to avoid reprocessing texel samples, or the ray tracing use of triangle pairs rather than single triangles, or even the A15/M2 SIMD shift-and-fill instructions (which seem very strange when you first see them, until its explained how they can be used to avoid having to reload data when performing large area image-processing operations like say a 5x5 filter).

There's also mentioned (but not in detail) that some compression is applied to vertices being temporarily stored (before subsequent tile-level fragmentation and pixel shading).

tesselation

The first stage of the graphics pipeline consists of, somehow, generating geometry. Initially this was just triangles; but we'd probably like to have curved surfaces in our images, so the Tessellation pipeline was born. The idea is to provide "patches" (fragments of curved surfaces parameterized in some way) which are converted by tessellator hardware into a collection of triangles.

Direct3D and OpenGL provided additional stages in their pipeline to control some aspects of tessellation, for example to decide the level of detail to which the tessellator should triangulate a surface – surfaces near the viewer should be broken up into many more triangles than surfaces far in the back-

ground.

(2016) <https://patents.google.com/patent/US20170358132A1> *System And Method For Tessellation In An Improved Graphics Pipeline* is essentially a reaction to these initial Tessellation pipeline designs, providing an API for Metal to achieve the same end results, but more efficiently.

One aspect of efficiency is omitting stages in the pipeline that a developer does not need to use, others include caching certain, slowly varying, properties from frame to frame rather than running the appropriate geometry-related shaders every frame, and giving the developer more control over geometry generation, like more ways to decide and indicate appropriate levels of detail for the tessellator.

A year later (2017) <https://patents.google.com/patent/US10621782B1> *Sub-patch techniques for graphics tessellation* gives details of how Apple's particular tessellator hardware goes about rendering a curved surface to a set of triangles with a particular level of detail.

Of course tessellation is just a specialized example of a more general problem, namely “I want to generate my own geometry, using the performance of the GPU, rather than dealing with a fixed geometry”. The current answer to that seems to be a mesh shading stage at the beginning of the pipeline in which can “unpack” (ie generate) geometry from however you have chosen to “compress” it (ie represent the full geometry).

We have meshlet hardware support with the M3, but going into details starts to stray way into graphics issues not hardware issues.

API

At this point let's try to connect the hardware to some aspects of the API. There are multiple GPU APIs, my concern right now is with the lowest level of Metal Compute programming. You can see elements of this as very similar to CUDA or ROC. On top of these lowest level API's more complex/generic API's can be constructed, but I will say little about them.

Also, for obvious reasons, much of what you read will put graphics foremost, but I'll be ignoring that as much as possible.

Before reading my stuff, you may want to read <https://therealmjp.github.io/posts/breaking-down-barriers-part-1-whats-a-barrier/>

This will act not only as a review but will give you something of an idea of just how primitive the GPU and its API's were as recently as 5 years ago.

At that point you may want to read at least the first few posts on <https://metalbyexample.com/2014/08/> which will explain the general concepts used in Metal programming.

Another article you may want to read is (2024) <https://xol.io/blah/death-to-shading-languages/> and more generally all the articles at <https://xol.io/blah/>. These articles give some insight into the “politics”

behind most graphics APIs and shading languages, and why they all seem so weak and ridiculous. (MSL is definitely superior in most dimensions, but hardly perfect. I suspect at least part of this has been a necessity for compatibility with graphics on the Intel Macs, and a desire to not make things too weird for “graphics programmers” all at once.

But I likewise suspect that the *ultimate* goal is the writing of “heterogeneous programs” that, more or less transparently, execute on both the CPU and GPU using essentially the same language; ideally something like blocks of sequential code that flow into blocks of parallel code and flow back to sequential. occam might be a *conceptual* model for this, though hardly a syntactic model.)

Finally, if you like reading patents:

OpenCL: Something that’s frequently forgotten is that Apple proposed, and did the early design work for OpenCL. You can see this playing out in a sequence of patents:

(2008) <https://patents.google.com/patent/US8225325B2> *Multi-dimensional thread grouping for multiple processors*

(2008) <https://patents.google.com/patent/US8286198B2> *Application programming interfaces for data parallel computing on multiple processors*

give the initial programming model. OpenCL1.0 came out in 2009.

The second also describes an OpenCL feature that never seems to have been much relevant in practice, namely write once, but execute the same code, on both the GPU and multiple CPUs; perhaps even simultaneously, to finish a single problem faster. Interestingly though that idea has essentially been abandoned for GPUs, it is what we see in the ML space, with Apple sending ML code to CPU, GPU, or ANE as it sees fit.

Meanwhile

(2010) <https://patents.google.com/patent/US20110285729A1> *Subbuffer objects*

describes the memory model once GPUs start having separate memory hardware (think VRAM or HBM).

This seems to have been implemented in OpenCL1.1 of mid 2010.

(2013) <https://patents.google.com/patent/US20150022538A1> *Enqueuing Kernels from Kernels on GPU/CPU*

describes a way, with very minor changes to the model, for kernels to be enqueued by executing kernels. I believe this was implemented in OpenCL2. This is presumably to match Kepler, which introduces this functionality in 2012.

This patent is for the (OpenCL) API side of things, it makes no claims as to how this is implemented; though later Metal Apple Silicon equivalents of this idea are concerned with implementation purely within the GPU, so that the CPU and driver are not involved. Even so, it shows a concern for iterative or irregular algorithms and provides means for these GPU-encoded kernels to synchronize and order relative to each other.

Pretty much the last patent from this OpenCL era, (2014) <https://patents.google.com/patent/US9442706B2> *Combining compute tasks for a graphics processing unit*, is of uncertain status; but it’s interesting at a conceptual level, so let’s discuss that.

Every introduction to GPU programming you read tells you to go crazy in allocating threads! Allocate one thread per pixel, one thread per element to be added or sorted or whatever, don't be shy! This is good advice when starting, to convert your mind from CPU-thinking to GPU-thinking. But at some point, once you start to care seriously about performance, it's time to re-evaluate.

It's usually (not always, but usually) the case that your code

- is primarily limited by memory performance (ie fewer load/stores to DRAM will speed it up a lot)
- involves some degree of data reuse.

Under these circumstances, it's generally better to think of your code not in terms of the minimal amount of work per thread, but in terms of trying to maximize reuse of data at every level, first within registers, then within the L1 or local storage, then within L2; rather than generating a vast number of thread blocks that each do minimal work, give each thread block a loop and size the amount of work in the loop to fit say L1 or local storage; then maybe allocate the number of thread blocks to match the size of L2. (Of course right now GPUs do a terrible job of making info like this available, so to some extent this is aspirational info to be matched as best possible to the practicalities of a range of targets.) OK, with this background in mind, the patent considers the idea of whether this can be done for you automatically. In other words you provide a minimal shader that handles, eg, one pixel, and the compiler (doing something like loop unrolling) crams four pixels of work into the call by replicating the code four times, or something like that. (Each time before the code is called, the registers that carry threadID and other such identifiers will of course also have to be updated...) The net result is that we have "right-sized" workgroups that try to balance the advantages of a large workgroup (lots of data reuse) against the advantages of small workgroups (easier to opportunistically launch these across multiple GPU cores).

This won't get us most of the data reuse that is most important, but will get us some reuse, along with also reducing the overhead of continually relaunching small workgroups. This is ultimately a compiler-type problem, and who knows how far Apple have taken it by now?

Metal: Around 2013 I guess Apple gave up on OpenCL for various reasons. (You can look into the politics of this and come to your own conclusions.) Regardless, the subsequent patents all seem essentially targeted at Metal.

(2014) <https://patents.google.com/patent/US20150348224A1> *Graphics Pipeline State Object And Model* gives a technical description of what Apple thought they were doing in designing Metal. Essentially the design philosophy can be summed up

- *what* is drawn is easy to change
- *how* it's drawn is expensive to change

The details of that *how* are encapsulated in the Pipeline State Object. Most obviously this incorporates the assembly code that will be passed to the GPU (and so building it requires compiling), but it may also include certain types of state change like setting special mode drawing mode registers.

Here's what I think is going on with this somewhat vague concept of a Pipeline State Object. Suppose we have a fragment shader that is responsible for coloring the pixels of some triangle by

texture mapping. And suppose the texture loads go outside the bounds of the texture map. What to do? You could imagine various options, for example one model is to clamp the address to the bounds of the texture, which is basically taking the outermost rectangle around the texture and repeating the colors of that boundary rectangle. Alternatively we could wraparound the texture, which gives us a repeating pattern, like a checkerboard tiling. Alternatively we could mirror the texture across the edges.

Now how might you implement these options?

One option is to have simple hardware where the general purpose *shader core* tests the addresses and does the appropriate thing. This means it's a simple matter of how the code was compiled as to what sort of texture wraparound is performed, and it seems reasonable, in such a world, to have an API call in your shader that looks something like `SampleTexture(texPtr, u, v, wrapMode)`.

Another option that might be used in more sophisticated hardware might be to have the *texture hardware* itself perform this address testing and appropriate remapping. Changing the wrap mode would then require changing some register in the texture hardware, and this might be mildly expensive (maybe have to wait for some sort of synchronization point after all work has drained from the texture unit, then change the register value) to very expensive (maybe the textures are preprocessed as they are loaded into the texture unit, and all this work has to be flushed when the texture wrapping mode changes?)

The problem is that the basic API call I described does not make it at all obvious that changing the `wrapMode` might be an expensive operation, it just looks like a trivial parameter to be set and changed whenever convenient.

So one solution to this is to move this sort of state out of “normal” draw function calls (where it can easily be changed) into a collection of setup/initialization calls, where it's clearer that setting up all this state is something of a hassle (and by implication, all the state you are allowed to change within “normal” draw function calls is cheap). The object that holds all this state, along with a block of code that is compiled to execute assuming this state, is called a Pipeline State Object.

From one GPU to another, the precise details of what state might need to be set up (eg via registers) and what might need to be handled in code (eg via compiling in texture addresses, and modifying that address when out of bounds) might change, as new features get added to newer GPUs. This is why we bundle together the concepts of the setup state and the code that executes on that state.

And since these transitions are somewhat expensive, rather than sequencing one expensive change after another, we do one “change the whole world” by swapping one Pipeline State Object for another, hopefully in such a way that overall this is cheaper than a sequence of multiple state changes, one register after another.

The ideal situation is we use one Pipeline State Object for the entire frame.

The more realistic situation is we have some number of Pipeline State Objects which we switch between, hopefully not too often.

The worst case is we create a new Pipeline State Object on the fly, because that may well invoke a

recompilation.

(I've used the example of texture sampling because I think it's easiest for non-graphics people to appreciate the point.

A more realistic example might be something like Multisample Anti-aliasing, where, if you understand MSAA, you will appreciate the point that changing the MSAA mode might involve some combination of recompiling the code [to execute a shader repeatedly at slightly different geometric locations] and modifying some registers [so that the results from the different shader invocations are accumulated and averaged in some way].)

Finally, when do we generate this code? Obviously the ideal solution is as close to the developer (and as far from the user) as possible! Ideally in the App Store, so the user just downloads the correct code for their target device.

If that's not feasible, then at the time of app installation.

If even that's not possible, then at the time of app *launch*.

Point is we absolutely do not want code being randomly recompiled during app *execution* (with implications for glitching the frame rate).

On the one hand this all sounds obvious when it's described this way. On the other hand, is it that obvious?

Certainly earlier APIs did not make this sort of distinction between state that was cheap vs expensive to change, with all sorts of horrible result for performance (both the driver having to keep checking that state had not changed from one call to the next, and the developer frequently making mistakes and using expensive state change calls because the API made such expensive calls too easy).

And earlier APIs likewise did not make strenuous efforts to avoid recompilation during execution (eg imagine you're using some 3D design app, and from a menu you change the wraparound mode of a texture; this could in these older APIs have triggered a shader recompile).

What Metal does in this respect is to some extent API-based (make it clear to the developer that these choices will result in expensive re-construction of a State Object) and to some extent OS-based (provide mechanism to allow finalized GPU code to be downloaded from the App store, or generated locally, either way cached in some known location for later reuse).

Along with this we have (2014) <https://patents.google.com/patent/US20150348225A1> *System and method for unified application programming interface and model* which cares a little more about the tools and developer environment to be used to create and describe the previously mentioned Pipeline State Objects.

(2015) <https://patents.google.com/patent/US20150347108A1> *Language, Function Library, And Compiler For Graphical And Non-Graphical Computation On A Graphical Processor Unit* expands the tools angle, most importantly by providing for libraries that can be called by kernels (which brings into play things like ABIs, the possibility of dynamic linking, and similar issues).

Finally (2016) <https://patents.google.com/patent/US10180825B2> *System and method for using uber-shader variants without preprocessing macros* continues the language/tools area, describing ways to

create families of Pipeline State Objects that are all very similar. You can think of this as more or less something like a GPU-side version of generics or templates, with the same sorts of issues as in Swift or C++ as to

- how to indicate what you want, on the developer side, and
- how to implement the result without code size explosion, on the compiler side.

Here's the problem we are trying to solve. Imagine a 3D authoring app (something like Blender) where you, as the artist, continually want to change and experiment with the material of some item in a complex scene. Conceptually this is easy to understand – the code submitted to the GPU contains a complex shader that can handle every possible variation of the material, and each time it runs (ie every frame) it looks at the settings for the material (opacity, reflectivity, whatever) and executes based on those settings.

The problem is that this sort of very general code is very expensive to execute, with a large number of if()s and table lookups and suchlike. Such code is usually called an *ubershader*.

So how can we improve things?

The option this patent provides works if you have only a few possible important variations (eg maybe there are four booleans that affect if-statements). Essentially what the compiler will do is the equivalent of what's called specialization in the context of C++ template compilation; it will compile a few different versions of the code based on the possible settings of the booleans, and then dispatch directly to the appropriate version. The bulk of the patent is concerned with issues like how you, the developer set this up and communicate your intentions; when you first compile the code, and then on every frame when you need to indicate the specific settings appropriate to the execution of this ubershader this time round.

In fact Apple has improved things somewhat since this patent, with a second alternative path now available.

You might ask: instead of using an ubershader why not just recompile some slightly modified code each frame?

The answer is that compiling a shader is expensive and may well cause the display to glitch for a few frames while the new shader is not available. But now we have a solution for that!

The alternative scheme (to be used if you can't capture and specialize the full complexity of your ubershader in a few compile-time variants) relies on the fact that Apple now offers *asynchronous* shader compilation, so the flow goes as follows:

- we start using the ubershader, while async compiling a variant optimized to the current material properties
- once the async compile is complete, we swap to using that optimized code
- then switch back to the ubershader and recompile when the material changes

This may sound like a lot of spinning, but what actually happens in a workflow is that you may have say 30 different materials in use. At any given time, 29 of them can be using the optimized compiled version, while only the material that you are actively tweaking [moving sliders and whatever, to change the material properties and see what it looks like] needs the full cost, every frame, of the

A slightly different take on things is (2016) <https://patents.google.com/patent/US9679346B2> *Graphics engine and environment for efficient real time rendering of graphics that are not pre-known*.

Here the issue of interest is drawing UI. The patent points out that, unlike games, where the developer tries to draw things in vaguely organized and thus approximately optimal fashion, UI consists of random independent entities updating their bits of UI on their own schedules.

The easiest way to handle this is by an “interpreter-like” first-come first-serve handling of the requests as they come in, but that means that we may constantly be toggling the state of the GPU (expensive) as we move from drawing one small control to the next.

The alternative is to look at the problem more like a compiler, ie to examine the entire scene-graph representing the UI, extract the various draw elements from that entire scene graph that are drawn using the same pipeline state, and bunch those together; thereby generating only a few pipeline state changes, and within each fixed pipeline state doing a lot more work across multiple controls.

At a rough level this is things like generating all blurs in a single pass, then doing all blending in a successor single pass.

This counts as a Metal issue in that the fine-grained control required to understand how best to sort the drawing calls, and ensure they happen relative to a single common pipeline state, depends on the availability of a low-level API like Metal, it’s not feasible in something like OpenGL.

Finally there are improvements that are essentially a collaboration between specifics of the hardware and specifics of the Metal API.

Consider an intermediate complexity graphics path, beyond the simplest basics. So assume, for example, that we want to fake shadows without using genuine ray tracing. Doing this involves something like running a compute shader that creates a temporary pixmap, loads some data into it (perhaps something like the Z position of each triangle), then uses that temporary pixmap along with some other data, maybe a texture of pre-rendered shadows, to modify the ultimate pixmap. There are multiple various other such temporary pixmaps that may be created along the way, perhaps to store the orientation of each triangle, or its material.

These temporary buffers go by the collective name of G-buffers.

Now, looking at the problem from an implementation point of view, these (conceptually anyway) represent storage that will be required during the construction of the frame. Thus our shader code needs to indicate the various buffers we want (eg the number of channels and bitdepth). In shader code, these are referred to as *buffers attachments*, and operating at a higher level than malloc, we indicate that we want them at a somewhat abstract level (eg number of channels and bitdepth) rather than allocating raw storage.

Now, how can we optimize this? It may be the case that a particular attachment is only ever used as *temporary storage*; we start a shader, we use the buffer to hold intermediate data that’s used to modify the ultimate image, and by the end of the shader we no longer care what happens to that buffer.

In that case, we can probably do without ever even actually allocating the buffer; instead we use some tile storage as a small window into the current region of interest buffer of this conceptual buffer. This is the concept of *memoryless render targets*. Memoryless means that there’s no allocation in the global (ie CPU) memory space, only the temporary allocation in GPU tile memory.

A similar sort of situation exists if you are implementing MSAA, where you execute your pixel

fragmentation into a higher resolution pixmap than the final pixmap, then combine the results of multiple pixels into a final resultant pixel; again the intermediate higher resolution pixmap is ephemeral and can be memoryless.

For this idea to be of value, we need tiled hardware (check), and we need a mechanism in the API (which Metal provides) whereby, along with all the other variables describing each buffer attachment, we also describe whether that attachment is memoryless.

This optimization is described in (2017) <https://patents.google.com/patent/US10825129B2> *Eliminating off screen passes using memoryless render target.*

The patent also clarifies one point I've continually wondered about; it states explicitly that (at least as of 2017...) tile data cannot be shared between render passes. Meaning that if pass A generates data that is used by pass B then never used again, the optimal route is to combine passes A and B into a single shader which can place that temporary data in a memoryless attachment. The patent actually gives an example of this, which somewhat corresponds to the earlier UI patent (*Graphics engine and environment for efficient real time rendering of graphics that are not pre-known*) updated to take advantage of memoryless attachments.

I have to admit there's a lot here that's still unclear to an outsider. I suspect that this is perhaps a place where the way I've been treating *kicks* and *shaders* as more or less the same thing breaks down. (Remember kicks are essentially synchronization boundaries such that, within a kick, there are no promises about automatic synchronization of data between cores.)

Kicks and shaders are more or less the same thing for compute, my primary interest, but it may be that for graphics the system is willing to pack multiple shaders into a single kick? So we decouple from shared memory (start of the kick) run multiple successive shaders (all flagged to operate on one tile/threadgroup successively, rather than running the shader to completion across the entire grid before the next starts) and only when we have run the full sequence of shaders against the full set of image tiles do we recouple to shared memory (end of kick)?

I believe (again may well be wrong!) that these map somewhat to Vulkan concepts, specifically a kick is somewhat like a Vulkan *Render Pass*, as a unit of activity that is decoupled from shared memory; and a Vulkan *Subpass* is like what I am calling a shader, a unit of work that acts on an image tile, with subpasses run sequentially over a single tile, and this sequence repeated for each tile, with no natural ability for a subpass acting on one tile to somehow co-ordinate with the same or a different subpass acting on a different tile.

This ability to limit all work to a single tile, one shader after another acting on that tile, is clearly the most desirable situation for both performance and energy. But it's not always feasible. When it's not feasible, then we have to

- separate work into distinct kicks (which essentially means first kick creates a *full* buffer of some sort; intermediate image(s) or whatever; as opposed to just a tile of data; then the next kick utilizes that buffer in some way); and we need to
- exploit the various DSID facilities to limit the performance impact and memory traffic caused by this

intermediate buffer.

BTW one of the names on this patent is Bartosz Ciechanowski. If this name is not familiar to you it should be! Take a look at his always delightful blog: <https://ciechanow.ski>

Another example of this sort of thing is (2017) <https://patents.google.com/patent/US10074210B1> *Punch-through techniques for graphics processing*. Consider the standard graphics pipeline: after vertex processing, triangles are binned to each tile which they will cover; after as many as possible are binned to a particular tile, they are sorted by depth order and those that are visible are drawn.

This works well because of locality (most triangles are small covering a fraction of a tile, and walking the geometry data structure also has spatial locality). And this ordering means we can avoid a lot of overdraw (ie the cost of texturing or shading a triangle that will be overdrawn by a later triangle)

However there is a problem when you encounter a triangle that has the potential to be non-opaque; you can't reliably make progress until you shade that triangle, and under normal conditions that means you then have to shade everything that could be below it (because of transparency) even if later all this work is covered by an opaque triangle. How can we fix that?

The patent describes splitting the traditional fragment shader into two distinct shaders, a *visibility* shader, and what we might call a *color* shader. The visibility shader is only run for triangles that are potentially non-opaque in some way (in other words drawing them requires drawing the triangle immediately below them in z-space), and only returns information about visibility, nothing else.

The workflow, then, is that fragments are accumulated in the tile as usual, except that fragments with an associated visibility shader have that shader called immediately.

Based on the opacity of the pixel, we can then store things as appropriate (track this translucent triangle *and* what is directly beneath it) while we continue to accumulate triangles.

If a subsequent opaque triangle then covers this translucent triangle, problem solved, we can drop the translucent triangle without ever bother to call its color shader, thus avoiding all the resultant costs of reading and applying textures or materials (and likewise for the triangle below the translucent triangle).

Bottom line is that the split fragment shader allows us to perform a (hopefully fast) visibility test separate from the more expensive operation of coloring pixels, in such a way that we can reduce the amount of overdraw.

Onwards, to Metal Shading Language!

The single most important thing you need to understand about MSL is the following thing that nobody ever explains:

When write a shader, you are writing a function with arguments. Now, how does the *call* of a function communicate the arguments to the *definition* of a function? The old-fashioned model, which is probably your baseline mental model, is arguments are passed by position, that is I define

```
void myFunction(type1 argument1, type2 argument 2, type3 argument3);
```

later I call this as `myFunction(a, b, c)`, and it's implicit by position that the `a` argument passed first will be "used" by the function as `argument1`, likewise for the second and third arguments.

Swift provides a little more flexibility by allowing us to give names to parameters when a function is called, as in

`myFunction(height: a, weight: b, temperature: c)`. If you're not familiar with Swift, a very brief overview of the relevant functionality appears here:

<https://www.hackingwithswift.com/quick-start/understanding-swift/why-does-swift-use-parameter-labels>

Now when you call a function that *you* defined, this is all fine; you can easily match up the function calls to the function definitions, and if necessary you can change both.

But think about this in the context of functions that are called by a runtime, as in your shader functions being called by the Graphics Runtime. How does the Graphics Runtime know what arguments your function will want, and in what order? And what to do when additional functionality is added to the Graphics Runtime?

The traditional answer to this, as in countless examples of OS driver code or Windows .COM APIs is something like

- the API defines what a callback function look like, eg there'll be a manual page saying "this is the prototype for a vertex shader".
- that API will include everything the designers can think of, so it requires passing around multiple (probably mostly unused) arguments.
- generally there will also appear one or two `void*` arguments, pointers to some structure. These serve two roles

+ there may be an OS structure that includes a few fields, with the hope that later, if necessary, the OS can tack more fields on the end of the struct. Older code should not be accessing those fields, while newly compiled code will be able to access them.

+ you, the developer, probably want to pass various data (arrays of vertices, textures, things like that) to your shaders, so you will define your own struct, pass it into every graphics call as a `void*`, and have the Graphics Runtime pass that `void*` on into every callback function (ie every shader you wrote).

Now obviously this is all a mess and not ideal. There's too much unnecessary work being done (passing unused arguments), there's far too little compile-time type testing being done, and it's unpleasant to code to.

MSL does better, but you need to understand the problem to understand the solution.

A basic metal shader might look something like

```
vertex MyVertexShader(const device Vertex *vertices [[buffer(0)]],
    constant Uniforms *uniforms [[buffer(1)]], uint vertexID [[vertex_id]]) {...}.
```

The `vertex` keyword says that this is a vertex shader (as opposed to various other shaders). As a

vertex shader, conceptually the task it that it is given a list of vertices (corners of triangles) and is expected to do something with each of them (most obviously multiply each by a matrix so as to position/orient them correctly from a 3-dimensional model space into a 3-dimensional world space). More specifically each lane of the GPU is given a distinct vertex on which to operate.

So this basic vertex shader (which, remember, you the developer will write) requires a few different arguments.

One is the list of vertices;

another is the specific ID of the vertex on which it is working (so that this lane knows which data to read from the vertex array);

another is the matrix by which we need to multiply the vertex to reposition it in space.

Rather than forcing a prototype that all vertex shaders have to use, MSL has the developer use the `[[something]]` syntax to indicate the *source* of a particular argument.

One type of source is something the graphics runtime knows, like the `vertexID` for a particular lane.

Another type of source is a resource, like a texture or a vertex array or a matrix. Instead of passing these around as part of a large untyped `void*` blob, Metal has the developer *bind* each type of resource during the pipeline building process.

So essentially in your *CPU* code you say

- I want slot 0 of the buffers to be this pointer (the `vertices*` pointer),
- I want slot 2 of the textures to be this pointer (some texture pointer),
- I want slot 1 of the uniforms (values that are constant across all lanes) to be this matrix, etc etc

Then later in your MSL code on the *GPU* side, you indicate that a particular argument slot should be connected to `buffer(0)` or `texture(2)` or whatever.

This is still clumsier than a perfect world, but is a great step forward over the earlier alternative.

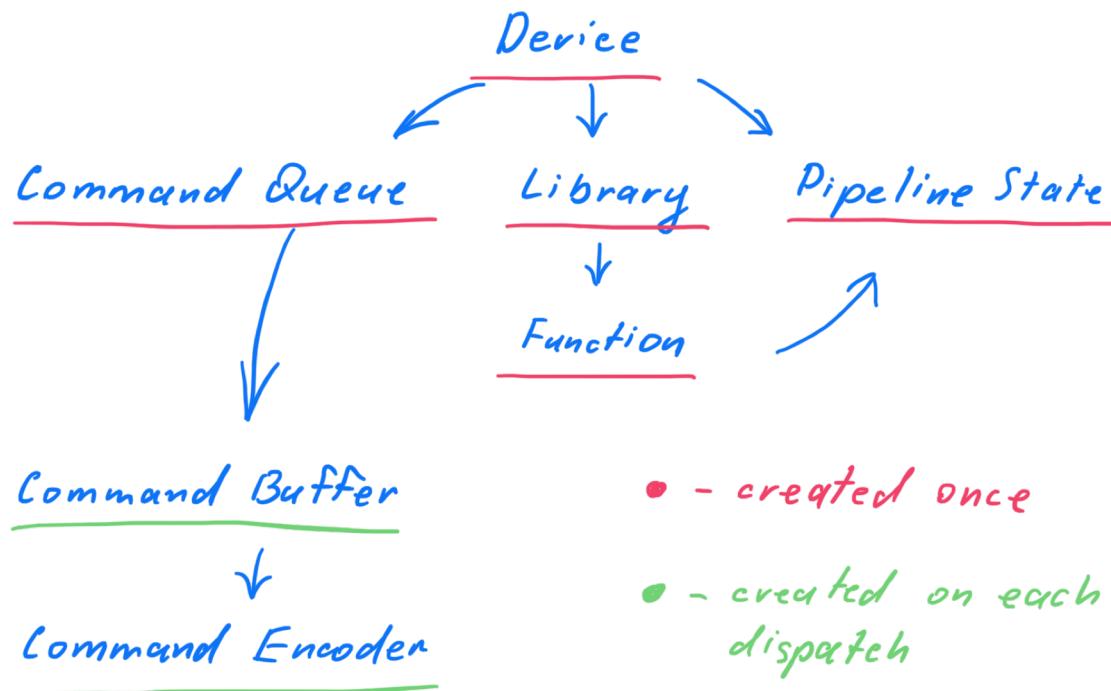
- Apple can add new functionality to MSL (and has done so repeatedly) by adding more of these `[[something]]` keywords to define new argument inputs to shaders.
- For the most part variables are now typed, so type-checking and similar mismatch detection can be performed.
- The Graphics Runtime has some knowledge of which arguments (textures, vertex arrays, etc) are being used when, and can optimize appropriately (insert some barriers automatically, prefetch data, attach DSIDs to these buffers, things like that).
- You get some degree (not much, but some) of the self-documentation of Swift in that many argument parameters somewhat clarify, via their `[[something]]` attribute exactly what their role is in a particular shader.

With this understanding in mind, you should now be able to read most MSL code. You may not understand the graphics aspect of the problem, but you should be able to appreciate the basic information flow from the CPU through the Graphics Runtime to the shader, and how that's conveyed by MSL.

At this point I remind you once again of <https://metalbyexample.com/2014/08/> which covers this material in a little more depth, though it soon veers off into graphics details.

To keep your head straight while reading all the above, you may find it useful to refer to this diagram below (from Eugene Bokhan's site)

(Unfortunately that site, <https://eugenebokhan.io>, now seems to be defunct; at least when I go there, the content is now gone!)



The other aspect to note, even if you are reading about these API's casually, is how every “resource” (ie

block of memory) is passed around the API in a rather structured way rather than the free-for-all of CPU programming. Among other things this discipline makes you work very hard if you want to do dumb things like alias and overlap resources, with all the resultant consequences for limiting optimization.

As I see it the most obvious way this matters is that every resource

- can be used as a synchronization mechanism (kernels that have no resources in common can execute in parallel, those that have read dependencies on a resource written by an earlier kernel will be held back until it's safe to launch them, without the developer having to write explicit synchronization code)
- has a clear lifetime
- *I think* this information is also used to move data around the system preemptively (eg start moving a texture or vertex list from DRAM to SLC or L2 in advance of when the texture will be needed by the next kernel to be scheduled). This would be an obvious optimization, but I've found nothing where Apple explicitly states it.

connections between MSL concepts and the hardware concepts we have seen

If you compare Metal code to the competition, eg CUDA, two obvious differences are

- Metal is much more explicit about the sorts of discipline and structure I mention above, which means the runtime/firmware can be a lot more helpful and a lot more aggressive in optimization.

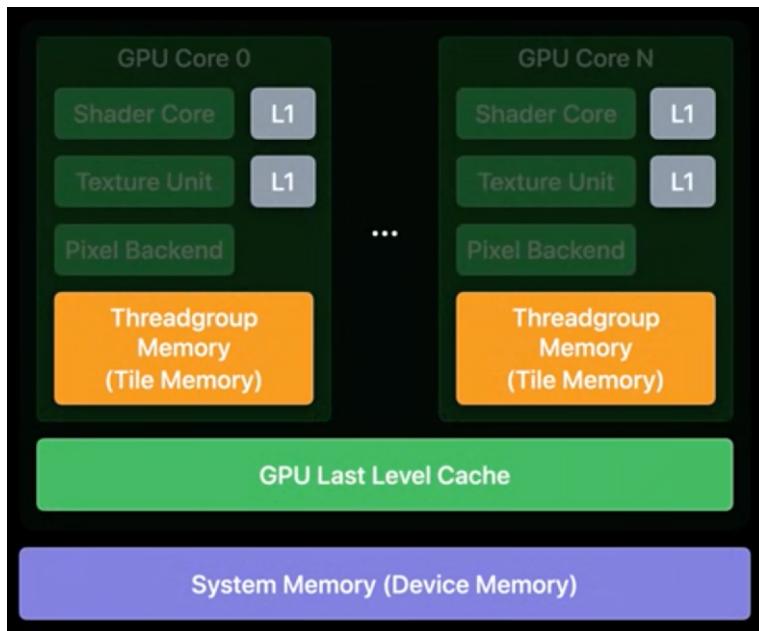
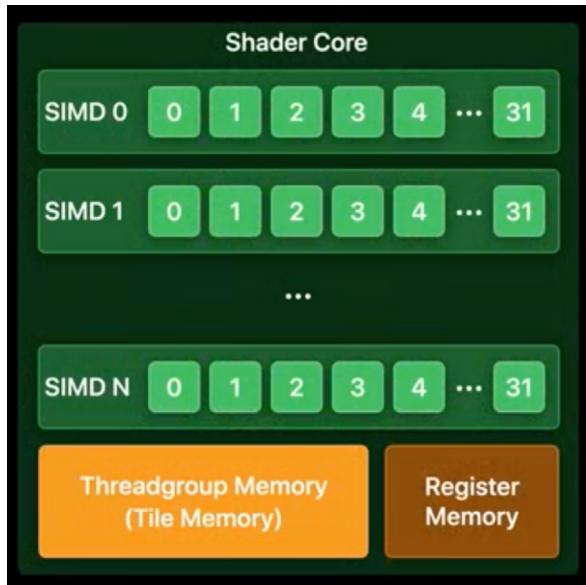
- Metal tries hard to remove dumb work from kernels.

CUDA kernels, for example, pretty much all begin with some boilerplate that converts the location-in-a-grid info provided by CUDA into integers that can be used as some sort of array index. Metal on the other hand provides pretty much every sort of such index you might want as a possible argument to the kernel, you just have to ask for what you need via the appropriate `[[something]]` annotation.

Similarly CUDA (and older Metal kernels) then have a bounds-checking step (since if your grid is 57 units wide, but the basic SIMD is 32 units wide, then by default you'll have threads running at locations 58..63 which need to be "switched off"). In recent Metal hardware (since the A11) the hardware itself can just mask off those lanes so that, unless you want to run your code on older hardware, this bounds-checking step can also be omitted.

We can see some elements of these various hardware elements in the API.

Essentially our hardware looks like



Within a core we have some number (currently 4) of SIMD's.

Local to the single lane of a SIMD is its registers. But (using the appropriate shuffle/permute commands) there is essentially just as fast access to the registers of the other three lanes in the same SIMD quad. (Remember how the four SIMDs share a common physical register block, with routing around that block that chooses exactly which register gets sent where, to handle dfdx and dfdy operations.)

Next up in locality is fast access, via shuffles/permutes to the other registers across the 32 lanes of the SIMD.

The SIMDs also all have access to core-local memory variously referred to as local/tile memory or the Local Image Buffer or Scratchpad.

Not shown is access to the Uniform registers, ie another pool of registers with values that are shared across all lanes.

The lowest level of the Metal API is the threadgroup. A threadgroup describes up to 1024 “execution lanes” that execute on one core.

So points are

- a threadgroup that is larger than 32 elements will execute in multiple SIMDs, perhaps across space (ie multiple SIMDs at once), perhaps across time, perhaps both (so four wide across all four SIMDs, taking eight time slots as we cycle through time).
- the first limitation on the size of a threadgroup is how many registers it uses. If it uses the maximum number of registers, then only six different “time slots” are available and the threadgroup covers the entire core. If it uses a minimal number of registers, then twenty four time slots are available, and the threadgroup will share the core with other threadgroups (ie will use “one third” of the core)
- the second limitation is the total local storage is 64kB, but one threadgroup can only access half of that, so if that's the limiter, you can see a threadgroup as using “half the core”

(Both the above describe the world prior to M3. Starting with M3, as we've described in great detail, there's a lot more flexibility in how registers are allocated and in the total amount of Threadgroup Local Storage.)

Ideally we want as little communication as possible between our SIMDs.

If we need to synchronize within a SIMD there are various techniques that can be used within MSL (Metal Shading Language) <https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf> based on the `simdgroup_barrier()` call, or various specialized functions with names like `simd_ballot()` or `simd_any()` which can be used if any (or all, or suchlike) lanes of a SIMD have achieved a certain goal.

If we need to synchronize within a threadgroup there is a `threadgroup_barrier()` call.

Thus within the context of a single threadgroup (meaning on a single core, but possibly across multiple SIMDs and multiple execution time slots) a single piece of MSL code (a single kernel) can have its up to 1024 threads all pause until they have all achieved a certain state, then continue.

Suppose, for example, that our 1024 threads are all supposed to load some data from RAM, compute something expensive (let's say a Bessel function or whatever) then add the 1024 results.

Your code would look something like

- we start the code by storing some special numbers used to calculate the function in some uniform registers
- each lane (based on its threadID) loads a value from an array in global address space. Global address space, also called device address space, is just the usual address space seen by the CPU and GPU.
- the thread will use its local registers to calculate products, sums, and whatever, occasionally perhaps multiplying in or adding in one of the uniform register values as special constants (π , e , or whatever)

- now once each thread has calculated its value, we need to sum them. But now we need to start to be careful because this involves sharing data across lanes/threads, and we need to be sure that data is ready.
- cheapest is sharing across a SIMD, so in principle the simplest version of what we might do is something like
 - + have them write out the data to local storage (something like `tmpArray[threadID]=value`)
 - + wait for all the lanes in the SIMD to reach the same point (because load/store may execute 8 lanes at a time, not 32 wide perfectly in sync...)
 - + ensure that the data written to local storage is visible to other SIMDs (is past various write queues and whatever)
 - + have one lane, say lane 0, read that data from the local storage in a loop and add it (something like `if (laneID==0) { . . . }`)

The part in this that you have to remember is the business of ensuring that all data written to the local storage is visible by all lanes.

This is done by the `simdgroup_barrier(mem_threadgroup)` call.

This is the simplest (but slowest) solution, but is a good example of the sort of case where you need a `simdgroup_barrier()`.

In actual code the better way to do this would be to perform the sum two elements at a time (eg $0+1\rightarrow 0$, $2+3\rightarrow 2$, ...) then two elements at a time ($0+2\rightarrow 0$, $4+6\rightarrow 4$, ..) and so on on, doing the job in log time. You would move the data between lanes using SIMD shuffles (which are essentially free for four neighboring elements, then cheap as we move across more lanes).

But of course the best way to do this is to realize that it's a reduction, that it's part of the MSL API, and so we can use the built-in `simd_sum()` operation.

OK, so that solves level one of the problem; we have the sum of 32 Bessel functions in lane 0 of a SIMD, and we want to add it to the 32 other such values in 31 other SIMDs (spread across space and time relative to ours).

Now there are no special helpers, so we have to use the techniques we described in our first SIMD solution.

- after each reduction store the SIMD sum value in lane 0 to an array in threadgroup local storage,

indexed by `[[simdgroup_index_in_threadgroup]]` or something similar.

- wait for all the threads in the threadgroup to perform this task, and ensure the written data is visible to all. This time we need to use a `threadgroup_barrier(mem_threadgroup)` as the barrier command
- have one SIMD load the 32 values (something like `if (SIMD_ID==0) { ... }`)
- execute `simd_sum()` and store it out to an address in global memory

I've been using some fairly obvious terms like laneID or threadID. In fact what MSL gives you is a few different IDs, like `thread_index_in_threadgroup`, from which you can construct exactly what you need. These IDs are called *Kernel Function Input Attributes*, and are described in the MSL manual.

The summary of what we have seen so far is essentially

- co-operation across a SIMD is usually cheap and easy. You mainly have to be careful if you can't find a built-in function or can't use shuffles, in which case each SIMD lane has to store its data in some common location (Scratchpad is the fastest so probably the best choice), and you need to use a barrier to ensure that every lane can see what every other lane has written (ie they all wait for all their write queues of all the other lanes to complete)
- co-operation across a threadgroup is reasonably cheap, will again require writing to a common location (generally Scratchpad) and will again require a barrier. Now the reason for the barrier is extremely obvious! Part of your threadgroup may be executing at a different time (in a different time slot of the barrel processor) from another part of the threadgroup, so we need to ensure that all the executions, across time and space, are done before we start to look at that communal data.

What if our problem is larger than a single threadgroup? Here's where things get trickier and multiple options open up.

For simplicity assume our problem is in fact just the size of two threadgroups.

Let's go in order of worst solution first.

If we submit the first half of the problem in one command buffer, the second half in a subsequent command buffer, we probably don't have to worry about anything. Command buffer barriers are really heavyweight, and between them everything written out to L1 by the first threadgroup should have been pushed out to L2 so that it's visible to all code and even to the CPU.

But heavyweight is another word for expensive and slow. Generally command buffers are used for the different frames being generated by graphics, so that after the frame buffer is done stuff more or less automatically finds its way to the display and/or the CPU with proper coherence.

But that's probably not a relevant way to split up the problem as far as we are concerned.

Alternatively we can simply make each thread in the threadgroup do twice as much work. After the first round of calculating Bessel functions and adding them up, we loop back, modify all the address calculations appropriately, load new function arguments, and repeat. This will certainly work, and won't require any new concepts; the problem is it only makes full use of one core of our GPU (and depending on the details, as we have described, it may only be making full use of only a third of the core).

So what we want to do is submit two threadgroups that can execute independently. We create a larger object called a grid (by Metal, sometimes a workgroup or similar by other APIs), and we submit the grid as a single task to Metal.

Metal will split the grid up into two threadgroups which may execute on two different cores, or on the same core in different time slots. Once you submit a grid all bets are off as to the location or relative sequencing of any one thread group relative to another. Note this point and never forget it...

This does not mean grids are useless! Return again to our Bessel function example. We can have the two threadgroups of our grid each, as their final task, write their sum of ($1024=32\times32$) Bessel function results to elements of a two-element long array in global address space, using an array index like `threadgroup_position_in_grid`.

Then when the grid is complete, the *CPU* can load the two values and add them.

(Storing to global address space will likely mean storing to GPU L1, which will be moved to GPU L2 at the end of the grid execution, and GPU L2 is coherent with CPU. Elements of this could be short-circuited by the compiler, eg store directly to L2, if it's smart enough.)

There are various variations on the above – instead of one large grid we could submit two threadgroups. As long as we give each threadgroup a different address at which to write the threadgroup's final sum, we don't care about the ordering.

But we still need the CPU to wait until the GPU has written the values to some (coherent) storage location before we can execute the CPU sum code. One option is to end our command encoding with a call like

```
[commandBuffer waitUntilCompleted];
```

which will do the job, but will also pause CPU execution until the GPU completes. We may want to submit our Bessel Sum asynchronously so that we can do other work while it happens.

In that case we need a third tier of synchronization known as a Metal Event: https://developer.apple.com/documentation/metal/resource_synchronization

Metal Events are no longer part of the *Metal Shading Language*, though they are part of the *Metal API*, because they don't happen on the GPU, they "happen" as a joint effort between the GPU and CPU.

Another way to say it is

- you write `simd_or threadgroup_barrier()` *inside* your compute shader (ie GPU) code,
- you write `memorybarrier()` to co-ordinate separate shaders *within a single command encoder*,
- you write `MTLFence()` *in your Metal CPU code* to co-ordinate shaders *in separate command encoders*,
- and you write Metal Event *inside your CPU code* to co-ordinate with the GPU.

I'm going into all this detail because, as far as I can tell, this is all somewhat unique to Metal.

Or, to be more precise, nVidia has now grown into these sorts of concepts (the ability to fine-grain co-ordination between different threadgroups) but they did so over multiple years, and adopting new terminology as each new level of co-ordination was added. As a consequence, people moving to Metal

from nVidia see what initially looks like a very “flat” system, just threadgroups, no concepts like “Cooperative Groups”, “Tiled Partitions”, “Domains”, and suchlike; and assume that only very coarse coordination is possible.

But that’s not correct; it’s just that Apple gets to the same point

- by scoping synchronization by *shared resources* rather than by *identifying groups of co-operating threads* AND
- by providing a second dimension of parallelism through independent Command Encoders.

Using grids with some cleanup on the CPU is easy and often a great solution, but let’s suppose we have a huge number of Bessel function values we want to sum, like say a million. Now we might want to write a first grid that executes a first pass, for each 1024 elements of a threadgroup writing out a sum value to an array in global space, then we submit a second grid (which fits into a single threadgroup) that loads the values from that array, adds them, and stores them back in global space.

This may not be as slow as you fear because the data from the first grid will be in the L2 ready for the second grid, and L2 latency, while not great, is also not terrible.

We are relying on the second grid running after the first grid to ensure that by the time the second grid runs, all writes by the first grid have been made globally visible (which is part of the programming contract). But this ease of use has some cost.

Alternatively, we get finer control of by writing a sequence of fine-grained threadgroups and

- use `memoryBarrier()` - if we encoded the threadgroups within a single Command Encoder, perhaps as a single large grid OR
- use `MTLFence()` if we encoded the threadgroups across multiple Command Encoders (unlikely for a large sum, but possible if we’re executing a set of heterogeneous tasks, like a neural net).

This is much like the barriers we described in MSL, except those wait *within* a single threadgroup, whereas

- `memoryBarriers` wait *across* threadgroups within a single Command Encoder, and
- `MTLFence()`s wait *across* threadgroups (within one, or across multiple) Command Encoders.

Metal supports a large number ($2^{15} = 32\,768$) of fences so you can use these all over the place to submit eg twenty threadgroups each with a separate fence, to be followed by twenty dependent threadgroups each waiting on a particular fence.

What do we get for this extra effort? This is something of a murky area! For each problem different issues may come into play.

Suppose for example that our problem naturally has the form:

task A and task B are independent (and fit into a threadgroup) right up till the end where we want to merge the results of A and B.

We might want to submit A and B as separate threadgroups, each setting an `MTLFence` when they end, along with a third threadgroup C that begins by waiting on the two fences, and then performing the merge.

This allows A and B to run on separate cores but with a transition to C that should be rather faster than if we ran these as two grids separated by an `MTLEvent`.

It’s also possible (though this is unclear) that the compiler and dispatch system heuristics will at

least try to run these three kernels in the fastest way possible; so that even though the system makes no promises, it could (if register usage allows it)

- run both kernels A and B on the same core,
- write out whatever data they wish to share with kernel C to both L1 and L2
- run kernel C on that same core, so that the shared data from A and B is mostly present in L1 without the cost of L2 access.

This sort of fine-grained submission of threadgroups rather than a large grid also allows you to match the order in which the threadgroups work to some pattern you may know about memory access that is not obvious.

For example if you submit threadgroups that work on data that is “close together” in some sense much of the data that is fetched and prefetched into the L2 and SLC may be shared across threadgroups; whereas if you submit a large grid, the system will choose who-knows-what ordering to submit the successive threadgroups, and that ordering may not provide optimal memory cache reuse, especially if there’s something non-obvious about your memory patterns.

And there are still other alternative solutions.

For example we could have each threadgroup, after calculating its local sum, instead of writing that sum to a separate address, uses `atomic_fetch_add_explicit()` to atomically add its local sum to a steadily growing global sum. Now the atomic add hardware does all the work of ensuring that writes that happen by different threadgroups all get appropriately serialized. This might even not be such a bad solution if most of the time is spent calculating the Bessel Functions; after the first few threadgroups keep colliding trying to perform the atomic sum, we’d expect that a separation in time develops, so that soon when any given threadgroup submits its sum every other active threadgroup is busy with calculation and there are no more collisions.

Yet another alternative is to ask “why am I trying so hard to do every calculation on a separate late?” Submit some small number of threadgroups (enough to saturate the GPU but no more, so maybe something like twelve or sixteen times the number of GPU cores) and have each one calculate multiple Bessel Functions and sum the value locally, then you only have one final round at the end where some co-ordination is necessary.

These issues are discussed in more detail in this thread <https://developer.apple.com/forums/thread/108416>; and in an nVidia document <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf>, which goes through sum of these options (of course in a CUDA context) showing the different costs.

Clearly this all remains somewhat unsatisfactory, at least in terms of documentation. I think it’s reasonable to conclude that

- who knows what happens on Intel Mac GPUs, but that’s not our interest!
- the current Apple documentation is an awful mess with multiples explanations somewhat out of date as new concepts (like `MTLDispatchType`) get added years after earlier documentation which assumed

only serial dispatch

- on Apple Silicon, there is the potential (from what we know of the hardware, and from what seems to be the case for the API) for things to work, mostly automatically and mostly efficiently, as long as you use the correct abstractions (like correctly declared, and non-aliased, buffers)
- but the Metal API/implementation (as opposed to hardware details) right now may not be as aggressive (ie efficient) as one might hope, simply because of the need for interop with Intel Macs; and that inefficiency may persist until those are no longer supported.

possibilities for the future (and abuse of existing hardware)

Especially if you are writing compute code rather than graphics code, there are all sort of interesting possibilities that might work utilizing resources in ways that Apple did not plan for. For example suppose that your code makes use of a mid-sized table, around say 4KB in size. Suppose that you loaded that table as a one dimensional texture and accessed it through the texture unit!?! You could even maybe get some useful computation from this by having the texture unit interpolate between values in the table. This gives you access to the texture cache as some additional storage (a few KB) without having to use up precious local memory or L1.

Another version of this, if it matches your code, might be to map some portion of local storage into ImageBlock storage, ie pretend that some portion of your local storage is a texture. The win this gives you is that some degree of the addressing required in accessing a two or three-dimensional struct will be handled for free when you access into this structure.

The most aggressive idea of all is to try to pass data in local storage from one threadgroup to another without writing it out to global storage (ie to L1). How can this be possible? Didn't we say that between different threadgroups anything goes in terms of whether they run on the same or different cores; and if you run on a different core you get a different block local storage with different values stored in it! Ah yes, well here's where we run into a mismatch between the hardware and the API. The compute kernel API's rules are as I described, but for the purposes of graphics Apple wants to dedicate that local storage (one tile, as in the TBDR, tile based deferred rendering, that defines Apple's 3D graphics design) to an entire graphics pipeline, with each successive stage of the pipeline able to work on what the earlier stage of the pipeline did. So, as far as API is concerned, one can in fact perform some compute in this tile, and then hand work over to the rest of the render pass which interacts with that tile, as described in https://developer.apple.com/documentation/metal/compute_passes/processing_a_texture_in_a_compute_function?language=objc

If you were willing to structure your problem as a graphics pipeline (as people used to do in the primitive days before modern compute shading languages) ...

This is all, of course, somewhat silly! But it does point out that there are multiple things Apple could do, most of them not especially hard, to make these GPUs more desirable for generic STEM. So far I've described

- some FP64 hardware (either at 1/16 or 1/8th the density of the FP32 hardware), at least enough to match the CPU+AMX FP64 resources, so that people have some incentive to start writing code, then

we see where that goes

- the use of the texture unit as a smart (interpolating) table lookup seems like it might be useful enough that it could be formalized in terms of MSL buffers and function calls, rather than having to make this weird sideways tangent into graphics to gain access to something we intend to use as a special function unit.

- “fat” compute threadgroups are allowed a large pool of register resources, but other possibilities are not opened up.

For example there seems no obvious reason why a threadgroup could not be given access to the full 64kB of local/tile memory on a core rather than restricting to 32kB.

A second possibility would be to allow larger threadgroups (up to 3072 in size) as long as they can still fit on a single core, to allow the basic unit of easy synchronization to be as large as possible.

This would be following the way nVidia has recently allowed for much larger “cheap synchronization units”, the equivalent of threadgroups that are larger than a single core.

- there might be value to providing a way to “chain” threadgroups together so that one threadgroup can inherit the local storage of the previous threadgroup. This already exists (and is encouraged) for the graphics pipeline where successive stages along the pipeline can inherit the local storage. The buzzword is “Persistent Threadgroup Storage”.

On the one hand this is less necessary for compute because you can (in principle) bundle all your compute work into a single large function that runs each of the stages of your compute, reusing the tile/local storage as necessary.

On the other hand, this forces the use of large single functions and works against attempts to bring modularization and libraries into the GPU compute...

On the third hand, this may all be moot given the new M3/M4 design. Maybe, going forward, if your problem does not naturally fit into Scratchpad limitations, you should just use global address space and standard synch primitives (including atomic variables). and just assume the L1 cache (as part of the pool of shared SRAM) will be as fast as Scratchpad, but able to overflow to L2 as necessary.

- another interesting (and perhaps feasible) way to get more value from a GPU is to allow shared resources. Suppose that we lay out two GPU cores so that the SRAM storage is back to back. It seems plausible that with just some minor additional wiring and muxes, we could reconfigure these two cores as a single core (shut down the other core’s registers and logic) seeing twice the local storage.

This sort of config (less compute available, but more fast storage) might be useful for at least some use cases...

- another way to look at this is that the graphics model, in a sense, is we have a tile of data (corresponding to a section of the screen) and we flow a succession of shaders against that tile. The starting point is the tile, and we say that we want these shaders, A..M, to be applied in order against that tile memory. So in a sense the data comes first, and we specify a sequence of functions that apply to it. One can imagine other data domains with the same property, a block of data that comes first, and we

want to apply a sequence of functions to that block, rather than starting with the function as the primary object.

Superficially grids look like this, but not really; grids are a set of shader functions arranged in space (ie given a set of spatial indices at launch time); they are not a block of data against which these shader functions execute...

- the other neat thing associated with ImageLocal/Tile memory is the R/W locks attached to each pixel allowing for certain types of fine-grained operation. Again these are exposed in the graphics API to some extent, for example allowing textures to be created then used (ie written then read) with the ordering enforced at a fine-grained pixel-by-pixel level (Raster Order Groups). Could this be used generically rather than just for pixel based code, eg by allowing the allocation of a generic Local buffer with some special OrderedLocal property?

what can we learn from the competition?

If we compare with the competition, what's Apple missing? Comparing against CUDA, as far as I can tell (and I may be wrong, some of these may be present on Apple, but I just couldn't find them) the list includes:

- funnel shifts. These are integer operations that concatenate two integers (ie 32bit values), shift by some amount, and extract 32 bits from the value. A simple use of this (which is provided by metal) is a bit rotation, but fancier use cases include bignum support.

- uniform datapath. This is nVidia's term for what I earlier described as a scalar datapath running in parallel with the 32 lanes of a SIMD, allowing occasional scalar (ie uniform across the 32 lanes, and only need to be performed once) operations.

- nVidia seem to have some more integrated language support for using the texture unit and storage as a way to hold lookup tables. For example Apple could do something like handle `Uniform<float> myTable[1024]` in this way; or, with a little more abuse of C++, something like `TexUniform<float, 1024, 0.0, 1.0> myInterpolatingTable` which compiles to something used as a function call, eg `bessel=myInterpolatingTable(x between 0.0 and 1.0)`.

- CUDA have support for *Cooperative Groups*, described in <https://developer.nvidia.com/blog/cuda-9-features-revealed/>

These don't seem to have very much new hardware content to them; much of what they offer can, even today, be simulated within Metal.

They allow either subgroups within a threadgroup to synchronize (something that could be achieved, at hassle, by splitting the kernel into separate kernels that each fully synchronize), or grids of multiple groups to synchronize within the MSL (ie CUDA) code rather than having to enforce the synchronization by encoding MTLFences and MTLEvents in the command stream.

But they are certainly convenient in terms of being able to express intent more clearly and simply! They also, by using the same primitive no matter what the hardware sync scale required (ie synching within a SIMD, within a core, across cores, or even across multiple GPUs) make it easier for developers.

- The same holds true for work nVidia has done to move beyond the limitations of lock-step lane

execution, so that certain algorithms that would otherwise deadlock on a SIMD design become possible (as described in the nVidia history Volta section).

code you may want to look at

If you are now fired up to try writing and benchmarking Metal code, let me suggest a few resources.

<https://github.com/philipturner/metal-benchmarks>

and

<https://github.com/ShoYamanishi/AppleNumericalComputing>

both provide various Metal benchmarks, and their code will give you a framework upon which you can build.

Given a framework, what to write and test?

I'd recommend you start by reading <https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf> (and honestly this is a great read even if you don't plan any coding). It considers about the simplest non-trivial GP-GPU problem, that of a very large reduction, and goes through many many successive refinements of the first, most obvious solution, each time picking up a factor of ~2 in performance.

You can then read a Metal version here: (2021) <https://betterprogramming.pub/optimizing-parallel-reduction-in-metal-for-apple-m1-8e8677b49b01> *Optimizing Parallel Reduction in Metal for Apple M1* which has many interesting asides. You'll see how MSL is much more like modern C++ than most shading languages, how many of the nVidia issues [eg banking] simply don't matter, and how the primary issue, the first question you should always ask, is "how do I optimize for memory traffic?" It's natural to think about optimizing arithmetic, but if the problem is large enough to require a GPU, memory will likely be your prime concern.

If you enjoyed that, the followups:

(2021) <https://betterprogramming.pub/efficient-parallel-prefix-sum-in-metal-for-apple-m1-9e60b974d62> *Efficient Parallel Prefix Sum in Metal for Apple M1* and

(2023) <https://betterprogramming.pub/memory-bandwidth-optimized-parallel-radix-sort-in-metal-for-apple-m1-and-beyond-4f4590cf5d3> *Memory Bandwidth Optimized Parallel Radix Sort in Metal for Apple M1 and Beyond*.

Like the reduction, prefix sum is still simple enough that your primary limit is simple memory bandwidth, and P-cores are performant enough that the CPU is probably the optimal tool.

But with sorting we finally hit a problem with enough internal compute that we're no longer primarily limited by memory bandwidth, and other design issues come into play.

