GPGPU Systems - Technical Report

Ayrton Chilibeck[1] and Danila Seliyaeu[1]

[1]Department of Computing Science, University of Alberta, Edmonton

December 12, 2024

Contents

Part I

Evolution of the GPGPU Architecture

0.1 Introduction - *Danila Seliayeu*

This portion of the report motivates and covers the architectural design of the GPGPU.

We start off introducing how graphics processing tasks shaped the hardware of the GPU and how the GPU evolved to cover more complex graphics processing tasks. The generality of the design naturally leads us to the exploration of how the GPGPU works at the architectural level. We then end this section by covering interesting developments in recent GPUs that relate to the discussed architectural content.

0.2 Graphics Pipeline

The graphics pipeline gained popularity thanks to the OpenGL API for graphics that released in 1989. It's popularity inspired hardware to dedicate individual units to its tasks, leading to many graphics accelerators implementing this pipeline in hardware [**mcclanahan2010history**].

The pipeline covered here has four primary stages ([**crow2004evolution**]):

- Application
- Geometry
- Rasterization
- Screen

with other minor stages scattered between.

This section will explain these stages and use them to narrate the evolution of the unprogrammable graphics accelerators commonplace in the 1980s into the general purpose hardware we are familiar with. More details can be found in the references, as well as in the Wikipedia page for the Graphics Pipeline which covers the same sections as this report [**wiki:Graphics˙pipeline**].

0.2.1 Application

We can understand the *Application* stage by starting from its output. Its output includes a set of vertices that encode information pertaining to objects to be rendered onto the screen and their transformations. The *Application* stage, then, naturally does the tasks necessary to produce or update the information encoded in these vertices. These tasks vary based on the application, but examples include handling physics in video games and animation.

The information encoded in vertices varies based on implementation. OpenGL's vertices include position, normal, and color ([**wiki:vertex**]).

0.2.2 Geometry

The next stage of the pipeline is the *Geometry* stage. This consists of a number of substages.

Transformation: Vertices represent objects but are unlikely to be in the correct position in space. These vertices are first transformed to where they should be in space. For a concrete example: if a programmer specifies a car's coordinates and rotation, the first part of the *Transformation* stage would be to perform transformations (typically via matrix multiplication) to get the car in the right place [[**nvidia256**]].

In addition to objects, an observer exists somewhere in space. The observer has a set of coordinates, a rotation, and a field of view that altogether dictate the view that should get rendered to the screen. In hardware, however, the observer's parameters are fixed with its position at the origin. Thus, we must transform our space to have the observer at this position. With this transformation, what the observer sees does not change. In this transformation, the 2D plane corresponding to what the observer sees is typically mapped to the x, y directions and the depth to z. Perspective transformations are also applied here to make what the observer sees in the x, y plane correspond to our perception of 3D space [**agoston2005transformations**].

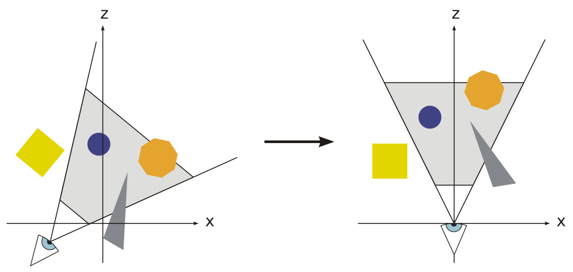

Figure 1: Transformation of the space to fit the observer.

An example of this can be seen in Figure 1 taken from [**wiki:Graphics˙pipeline**].

Lighting: The *Lighting* stage modifies the color values of our transformed vertices. Lighting information is given to us by the *Application* stage. This includes information about light sources, like spot lights and ambient light, including their strength and orientation. This information is largely encoded in matrices and the color values of vertices can be modified with a series of matrix multiplications[[**nvidia256**]].

Primitive Assembly: This stage groups our vertices into triangles and other primitives. Triangles are the simplest polygon that we can combine to create other polygons, making them convenient to work with ([**wiki:overview**], [**scratchapixelRasterization**]).

Clipping and Culling: The world our space represents may have objects that the observer can't see. The primitives composing these objects are entirely removed in what is called **culling** [**graphicscompendiumGraphicsCompendium**].

Some objects overlap with the boundaries of the observer's field of view. The triangles making up these objects are reshaped — converted into smaller traingles — to fit within the boundaries. This process is called **clipping** [**graphicscompendiumGraphicsCompendium**].

An example of clipping and culling can be found in Figure 2 taken from [**wiki:Graphics˙pipeline**].

0.2.3 Rasterization

After we pass the *Geometry* stage, we have a set of triangles. This stage converts those triangles into fragments.

Fragments can be generated through the following algorithm: First, iterate through all triangles.

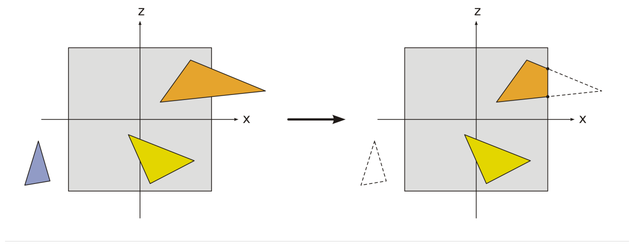

Figure 2: Examples of clipping (of the blue triangle) and culling (of the orange triangle).

Next, iterate over all pixels. If a triangle overlaps with a pixel, create a fragment with those pixel coordinates belonging to the triangle [**scratchapixelRasterization**].

Fragments differ from pixels in that there can be multiple fragments corresponding to a given pixel's position on the screen. In terms of storing color information, they are the same.

Fragments are used in later minor stages of the pipeline.

0.2.4 Screen

The pipeline ends with fragments rendered to the screen as pixels. Rendering happens through writing colour data to a special region in memory mapped to the screen.

0.3 Early Evolution

Motivated by the popularity of the graphics pipepline, graphics accelerators in the 1980s and 1990s largely implemented a fixed pipeline in hardware ([**mcclanahan2010history**]). This section traces the evolution of this fixed design into one that supports general computation. This section focuses primarily on Nvidia and their role during this time.

0.3.1 Graphics Processing Unit

With the design of graphics accelerators heavily coupled to the pipeline, one critical difference between accelerators was how much of the graphics pipeline did the accelerator actually support.

Until 1999, graphics accelerators did not support the acceleration of *Transformation* and *Lighting* sub-stages (of the *Geometry stage*) of the pipeline. This meant that the CPU would do the *Application* stage and the the *Transformation* and *Lighting* sub-stages before passing the resulting vertices to the accelerator to do the rest ([**mcclanahan2010history**]).

Nvidia was the first company to put the entire *Geometry* stage onto the accelerator, allowing the CPU to focus on the *Application* stage [**nvidia256**]. This meant more high fidelity simulation.

Nvidia marketed this as a graphics processing unit, popularizing the term.

0.3.2 Programmability with Shaders

Processors started to support programmability through what were called shaders. In hardware, shader units supported code that could be written by the programmer to specify desired fuctionality at given parts of the pipeline ([**mcclanahan2010history**]).

Vertex shaders replaced the *Transformation* and *Lighting* parts of the graphics pipeline. The programmer could specify the specific transformations and lighting operations that they wanted performed, as well as other operations ([**lsu**]).

Fragment shaders were added after the *Rasterization* stage. These enabled the programmer to perform transformations on the fragments before they were rendered to the screen ([**lsu**]).

Together, these shaders enabled more advanced graphical features to be implemented.

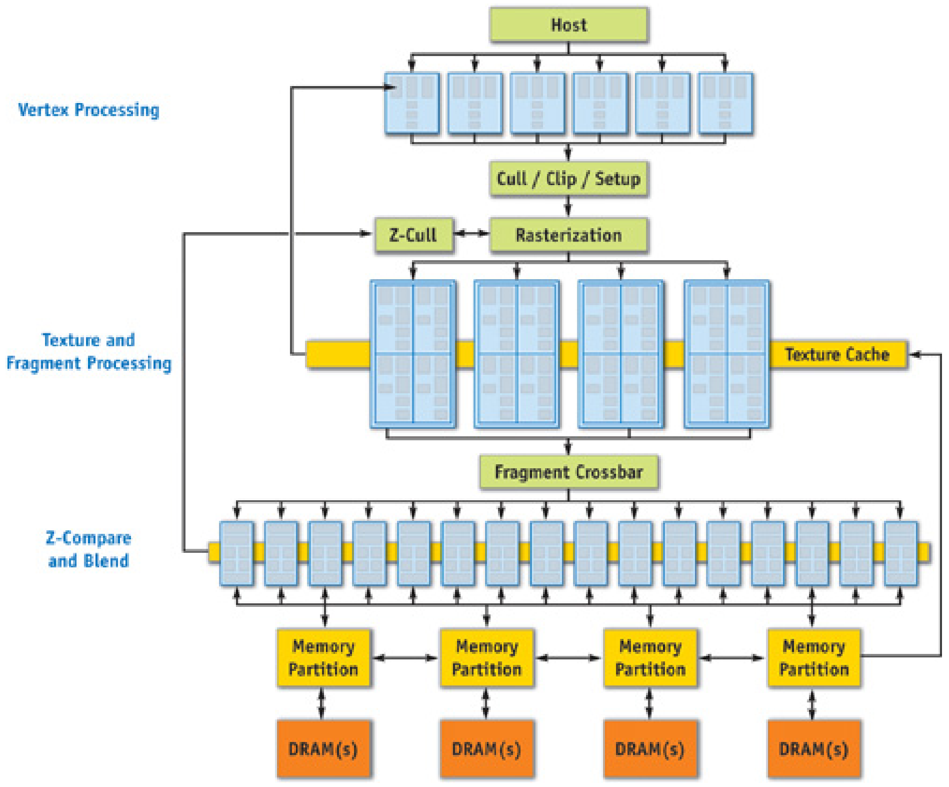

Figure 3: Diagram of the GeForce 6

An example architecture that implemented shaders is in Figure 3 [**nvidiaChapterGeForce**]. Note the Vertex Processing (vertex shaders) and Texture and Fragment Processing (fragment shaders) using different processing units.

0.3.3 Unified Shader Architecture

The unified architecture replaced bespoke hardware for different shaders with a single processor capable of doing all relevant shader tasks. This meant that computations for all of these different tasks could be performed using the same ISA. This ISA thus had to be general, paving the way for the use of the GPU for applications beyond graphics [**parojModernUnification**].

Figure 4 shows Nvidia's GeForce 8 architecture [**mcclanahan2010history**]. This architecture has a number of streaming multiprocessors (SMs) with individual streaming processor cores (SPs). A controller manages what shaders execute on these SMs, allowing their output to feed back into them.

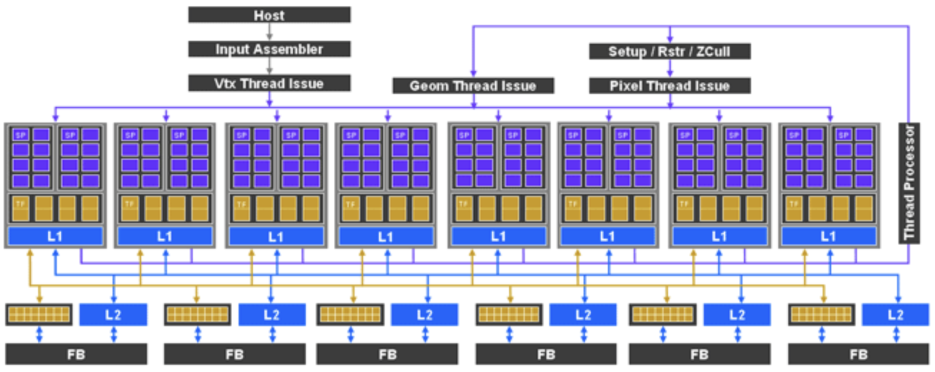

Figure 4: Diagram of the GeForce 8.

0.3.4 The General Purpose GPU

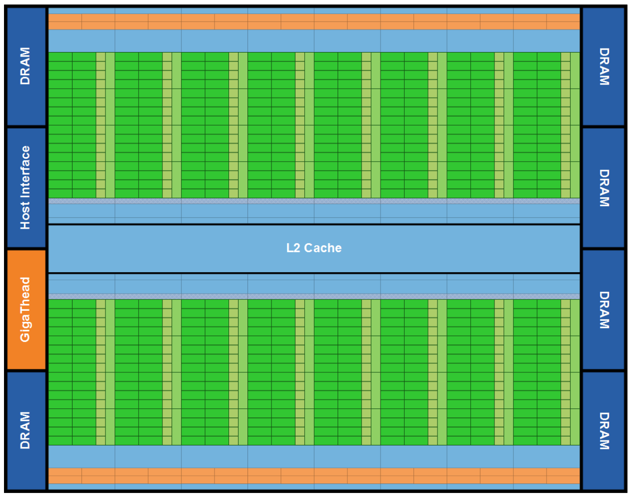

Figure 5: Diagram of the entire Fermi architecture.

Nvidia had an interest in general purpose GPU computing at least as early as during the development of the GeForce 8 series ([**dally2021evolution**]). Fermi represented their first card marketed as a general purpose accelerator ([**nvidiafermi**]). The general architecture of the chip can be seen in Figure 5 and a single multiprocessor core can be seen in Figure 6 ([**nvidiafermi**]).

0.4 GPU Terminology

To understand the architecture of a GPGPU, we need to define some terms first. This section will define various GPU related terms using CUDA examples and their Nvidia names.

0.4.1 Programming Concepts

```
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
```

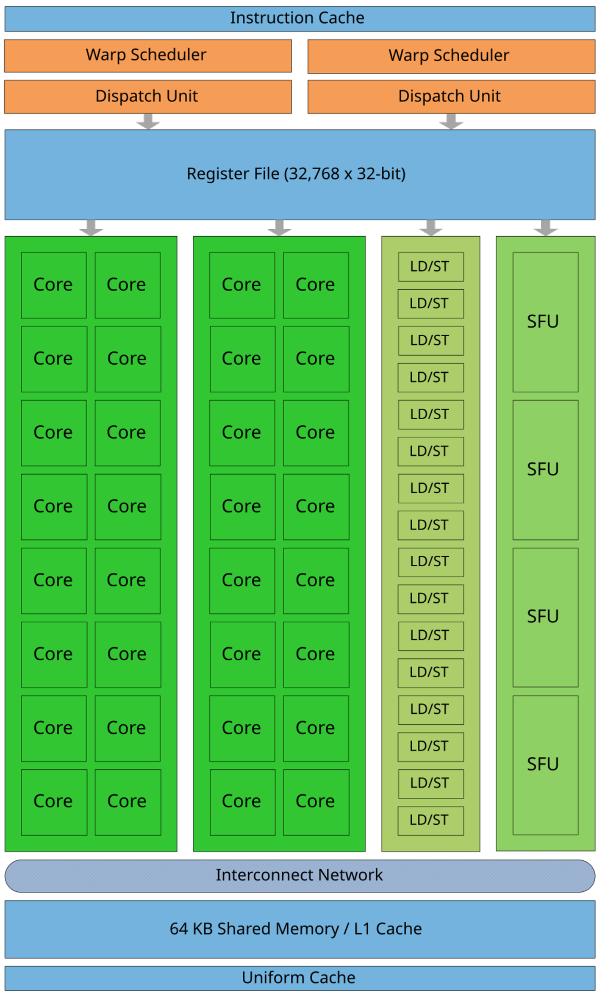

Figure 6: Diagram of a single Fermi SM

```
    C[i] = A[i] + B[i];
}
```

Listing 1: A simple CUDA example.

Threads: GPU threads can be thought of in the same way one might think of CPU threads. They perform some fraction of computation [**nvidiaCUDAProgramming**].

The code in Listing 1 shows us how we can use threads to add two vectors, A and B, forming a vector C. *__global__* indicates that this code is to be run on the GPU ([**nvidiaCUDAProgramming**]).

If we were to have 1000 elements in these vectors, we would do this by spawning 1000 threads, each of which would get assigned a $threadIdx.x$ from 0 to 999. Each thread would perform the operation corresponding to its id.

The assignment of these values to threads is largely arbitrary in that we could shape them to be something else. CUDA supports $threadIdx.y$ and $threadIdx.z$ in addition to $threadIdx.x$, allowing us to organize our threads into three dimensions when we spawn them.

Blocks: We spawn threads in groups called blocks. The sizes and dimensoins of these blocks dictate the number of threads and how they are assigned *threadIdx*s. We can spawn multiple blocks — which we might do if we want to, say, do vector addition on many pairs of vectors. Blocks are organized in grids [**nvidiaCUDAProgramming**].

0.4.2 Hardware Concepts

Computation on GPUs is organized with many streaming multiprocessors (SMs). Blocks are assigned to SMs. SMs run one block at a time but can have many more assigned to them ([**aamodt2018general**], [**nvidiaCUDAProgramming**]).

SMs are composed of many small cores called streaming processors (SPs) or CUDA cores. Threads run on these cores ([**nvidiaCUDAProgramming**]).

The previous section showed SMs with multiple warp schedulers scheduling compute across multiple sets of SPs. We will revisit this modern organization of compute later.

For now, we can imagine that an SM has 32 SPs that all execute their threads in lock-step. These threads are grouped and refered to as **warps** ([**aamodt2018general**]).

0.5 Three-Loop Approximation

One convenient way of understanding how the GPGPU works is through looking at its three main scheduling loops that run within every SM. This is the approach taken by [**aamodt2018general**]. The textbook builds the architecture up, covering one scheduler at a time.

This section will use the three approximations introduced in the textbook but leave the treatment of warp divergence for last. The reader is encouraged to refer to the textbook for more details.

0.5.1 One-Loop Approximation

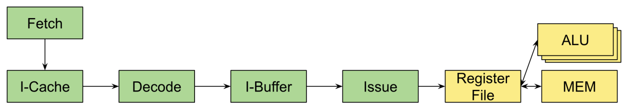

Figure 7: One-Loop Approximation diagram.

The first loop is the scheduler that fetches instructions.

Refer to Figure 7 for a high level overview of the pipeline at this point. The Fetch unit sends an instruction from the Instruction Cache (I-Cache) to get Decoded before it gets saved to the Instruction Buffer (I-Buffer).

In this approximation, we can have one instruction in flight at a time for a warp and we fetch instructions using the warp's program counter corresponding to this instruction.

While the CPU issues many instructions at once to allow for latency hiding, the GPU hides a significant amount of latency by Issuing instructions for different warps at different cycles. Specifically, if we have a warp that is performing a memory access that missed in cache, we can swap to another warp whose next instruction uses an ALU, hiding the latency.

0.5.2 Two-Loop Approximation

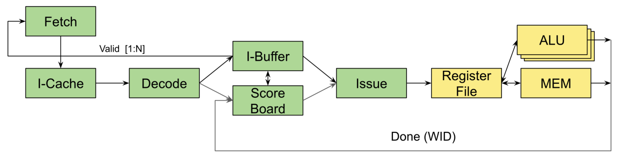

Figure 8: Two-Loop Approximation diagram.

The second loop is the scheduler that chooses which instructions to issue. Refer to Figure 8 for a high level overview of the pipeline at this point.

In order to support swapping between warps, we need to save the values in the registers for each warp when we context switch. This means that our register file must have more space and be physically larger for every warp we support. As a result, this limits us in regards to how many warps we can allocate to an SM. This means that past a certain point, we may need other means of hiding latency.

Another way we can hide latency is through instruction level parallelism with multiple instructions from a warp executing at the same time. Doing this naively can result in write-after-read hazards. As a result, we need some means of avoiding hazards. Since warps execute instructions in-order, a natural solution is the scoreboard.

In in-order CPUs, scoreboards simply have a ready bit for each register. This doesn't work for GPUs, though: we have over a thousand registers we would need to track, occupying significant space.

One solution is the following: for each warp, keep a list of the destination registers of all in-flight instructions. When an instruction gets added to the I-Buffer, it can check whether its operand registers match those in this list and save to its row in the I-Buffer a bit vector whose bits indicate whether or not an operand is ready (set to 1) in the sense that it will not be written to by any in-flight instructions.

In the write-back stage, the instructions update the list of registers in the scoreboard, removing the one they write to. They also go through all instructions in the IB for their warp and update the ready bits for those instructions whose operand registers match its destination register.

Instructions with all bits in their bit vectors set to 1 can be scheduled in-order by the scheduler. These then add their destination register to the scoreboard and begin execution.

0.5.3 Three-Loop Approximation

The third loop is the scheduler that optimizes accesses to banked register files.

A single-ported register file can be read from or written to by one operand on a given cycle. If we want to increase the number of accesses, we can increase the number of ports, but this requires a lot of area.

An alternative approach to enable more concurrent accesses is banking. This consists of splitting our register file into smaller register files called banks. Each bank holds different data and has its own read/write port. This means that each bank could be accessed in one cycle by different operands. Accesses for operands pass through an arbitor that routes them to the correct bank.

Registers are mapped to banks in a swizzled fashion. A naive mapping of a register to bank $r\#\ \%\ NUM\_BANKS$ would assign registers 0 of all warps to bank 0, registers 1 to bank 1, and so on. If we try to fetch the same register for two different warps, then, we would have to stall because we would have two accesses to a single bank in one cycle.

Swizzling fixes this issue. It makes the warp number affect our mapping. Thus, a given register is now assigned to bank $(r\# + W\#)\ \%\ NUM\_BANKS$. This way, the same register on different warps maps to different banks.

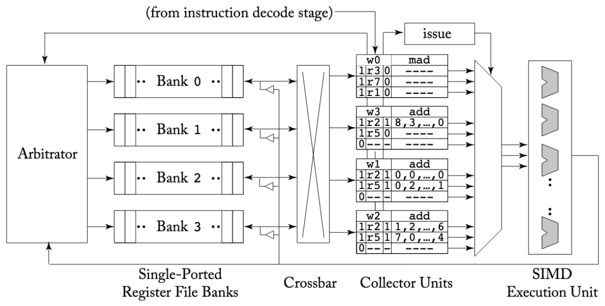

Figure 9: Hardware for accessing the register file using the Operand Collector.

To schedule operand accesses, the Operand Collector is used. This is a hardware unit that has a slot for each instruction. These schedule register bank accesses to minimize conflicts. Figure 9 shows how the banked registers could be structured with an Operand Collector.

Refer to [**aamodt2018general**] pages 35 to 40 for detailed examples that illustrate the benefits of using the Operand Collector and bank swizzling.

0.5.4 Warp Divergence

This approximation has omitted handling warp divergence for the sake of simplicity. We can have control flow in CUDA code. An example can be seen in Figure ??.

This is the same code as before, except for the fact that the size of the vector is passed as an argument, N. This could be specified by a user, unlike the number of threads in a block, which is hardcoded. We assume that N is less than the number of threads in this example.

One question that might arise is what if N is 16? Warps consist of 32 threads executing in lock-step. How do we handle this?

The answer is predication: we use a mask (the SIMT Mask) to control which threads in a warp should be active (set to 1) and inactive (set to 0) at a given point in the program's execution.

The SIMT Mask is managed by the SIMT Stack.

0.5.5 Full Architecture

0.5.6 Three-Loop Approximation

With the SIMT Stack, we can assemble a diagram of the entire architecture. This can be seen in Figure 10

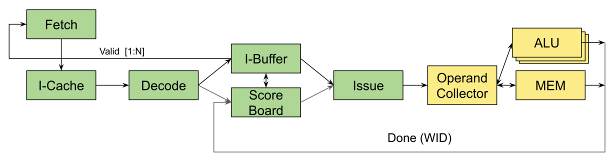

Figure 10: Three-Loop Approximation diagram with SIMT Stack for handling warp divergence.

0.6 Modern GPGPUs

This section will cover recent developments in GPGPUs.

0.6.1 Reality of Warps

Recall the Fermi SM diagram in Figure ??. We have two sets of 16 SPs corresponding two two warp schedulers, giving us 16 SPs per warp.

Imagining warps as 32 cores executing in lock-step is useful in explaining things, but it isn't realistic. In reality, warp execution takes two cycles as shown in Figure ??.

0.6.2 Recent Developments

Kepler: This is the first Nvidia GPU to introduce a compiler generate control instruction. This is inserted every 7 instructions. One byte of the 64-bit instruction indicates that this is a control instruction. The other 7 bytes encode information for the next 7 instructions [**chipsandcheeseInsideKepler**].

This control instruction encodes latency information, preventing write-after-read hazards from occuring between fixed-latency instructions [**chipsandcheeseInsideKepler**].

These instructions still modify a central scoreboard that the scheduler that issues instructions looks to [**chipsandcheeseInsideKepler**] [**adalbert2022pastis**].

In addition, these instructions encode whether or not instructions can be dual-issued [**chipsandcheeseInsideKeple**

Maxwell: This architecture increases the frequency of the control word, inserting it every 3 instructions instead of every 7.

Maxwell control instructions include information pertaining to variable latency instructions. This works by assigning destination registers of these instructions to a barrier in hardware. Instructions that then have those registers as operands must wait until the barriers clear up. This means that Maxwell needs less scoreboarding hardware [**chipsandcheeseMaxwellNvidias**].

A one-bit yield flag is also introduced in this architecture. This instructs the scheduler to prefer to switch execution to another warp instead of continuing instructions from a single warp ([**githubControlCodes**], [**jia2018dissecting**]). The hardware still makes most scheduling decisions on its own but there are situations where this bit can lead to performance improvements [**githubControlCodes**].

This architecture also introduced an operand reuse cache [**chipsandcheeseMaxwellNvidias**]. In this architecture, registers are assigned to banks naively (register number modulo 4, the number of banks). What allows this to work is the reuse cache. The cache has entries for 4 operands. When an issuing an instruction, a hardware unit checks the corresponding part of the control instruction to see whether or not flags indicating that some registers will be used again are set. If so, it can save those registers in the reuse cache. Future uses of those registers in the same operand slot can access the value from the reuse cache, avoiding having to access the register file, reducing bank conflicts ([**jia2018dissecting**], [**githubSGEMM**], [**chipsandcheeseMaxwellNvidias**]).

Pascal: Pascal introduces a new way of handling warp divergence.

Volta: Volta increases the instruction word from 64 bits to 128 bits. It also starts to include control information within instructions instead of using a control word[**jia2018dissecting**].

This architecture also introduces the Tensor Core [**sun2022dissecting**].

Part II

GPU Memory Systems

0.7 Introduction - *Ayrton Chilibeck*

My portion of this report focused on the GPU memory system. Through the development of this report, the presentation and the GPU assignment I learned that the struggles of using the GPU memory hierarchy effectively is vastly different than the use of the CPU hierarchy. This part of the document will address those differences as well as clear up some fundamental questions about how memory works in the first place.

We will begin by addressing the construction of memory cells at a transistor level. This will encompass discussion around both the structure of DRAM and SRAM as well as the underlying structure of the transistor itself. We then proceed to the difference in purpose between the GPU and CPU memory systems, as well as the tradeoffs associated with either one. We will discuss differences in design principles underlying both CPU and GPU compute units as well as the varying design of DRAM systems associated with each system. We then discuss the architecture of the GPGPU according to the latest documentation available from both NVIDIA and AMD. This is accompanied by an exploration of the sub-structures contained within each discrete unit of the GPU and how they interact. We also discuss novel architecture decisions and advances in the literature over the last several years and follow this by several worked examples of memory accesses in the GPU memory system. The next section covers the need for memory coherence at the local memory level in certain scientific applications and cache coherence protocols required for such computations. Finally we address novel architectural problems related to the development of heterogenous systems (where both the CPU and GPU share a single die) and how such problems are overcome in the literature and the (admittedly sparse) technical documentation available from AMD, Intel, NVIDIA and Qualcomm.

The most impactful part of this assignment was the chance to spend a significant amount of time refining my understanding of memory systems. I hope to impart some of what I learned in this document and expose certain areas of interest for the reader to explore further.

0.8 The Electronic Basis for Memory

In our studies of computer science at the University of Alberta, we rarely mention the underlying electronics that form the basis of electronic logic systems, and hence computers. Despite our high level abstractions away from silicon when learning how to write code, a robust understanding of the underlying principles of computing systems encourages students to think about the physical limitations of our current systems and why we design our systems as we do. This section will scratch the surface of electrical engineering and provide the reader with the essential background to determine why certain artifacts, like CAS latency, appear in the analysis of the memory subsystem.

0.8.1 The Transistor and Capacitor

When studying memory systems, there are two fundamental structures we need to understand: the transistor and the capacitor. For our purposes it suffices to think of the capacitor as a battery and the transistor as a switch, the following section is an elaboration on those simplifications.

Beginning with the capacitor, a capacitor accumulates and stores charge as the buildup of a potential difference between two surfaces insulated from each other CITE. They serve as the physical storage for a single bit in conjunction with a transistor and a ground. Charging a capacitor is a relatively slow operation (relative to the modification of an SRAM cell, as we will discuss in

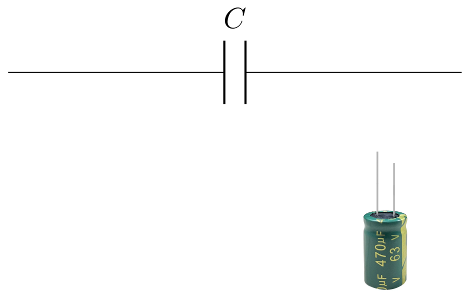

Figure 11: A Capacitor.

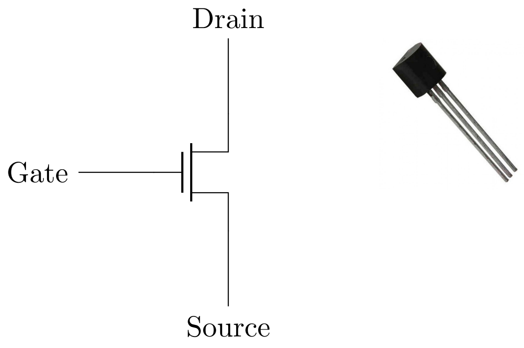

Figure 12: A Transistor.

a moment), but the die area of a capacitor and single transistor is significantly less than the die area required for a single cell of SRAM, thus allowing us to produce large capacity DRAM cheaply CITE.

The transistor is our next target of study. Simplistically, the transistor is a switch. It consists of three connections: the source, the gate and the drain. The gate serves as the switching mechanism: when we apply current to the gate, current can flow freely between the source and the drain. When no current is applied to the gate, no electricity can flow from the source and drain. The transistors used in hobby projects are often designed to be one-way transistors, however the transistors used in common DRAM systems are designed to allow current to flow both in and out with the same resistance CITE. Current transistor technology relies on the MOSFET paradigm, though Intel's upcoming 16 Å process is rumored to use RibbonFET/FINFET technologies CITE.

The transistor and capacitor form the basis for our current memory technologies.

0.8.2 Storing a Single Bit

We can combine the transistor and the capacitor to form a single DRAM cell as illustrated in figure 13. This unit allows us to store a single bit through the following process:

1. Apply current to the bit line
2. Apply current to the gate, resulting in a potential difference letting the capacitor charge
3. Eliminating power to the gate, effectively locking the power into the capacitor

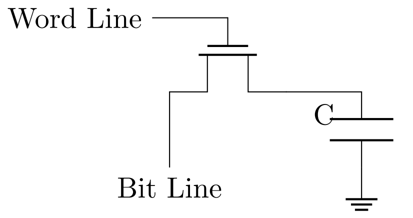

Figure 13: A single DRAM cell.

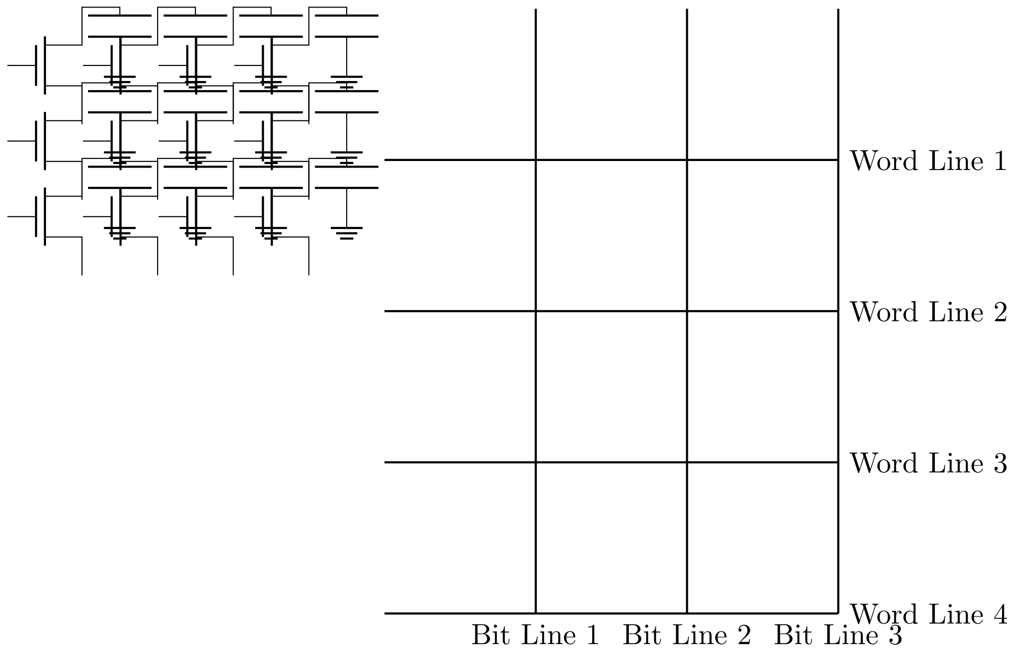

Figure 14: A DRAM Array.

A capacitor of this size loses its charge quickly, resulting in the need for periodic refresh of the entire DRAM array. This refresh essentially consists of reading and rewriting all of the elements in the array in order to maintain their charge.

We can chain the individual bit storage mechanisms together to form word lines and group those together to form the full DRAM array as shown in figure 14.