You know I haven't eaten since 6 o'clock this morning, and that was half a cream cheese bagel. And it wasn't even real cream cheese, it was light cream cheese! Now you want me to run off and do

# Alternative Processor Architectures

ENGN8537

Embedded Systems

# Overview

- Pipelining Review
- SIMD
- Stream Processors
- GPU

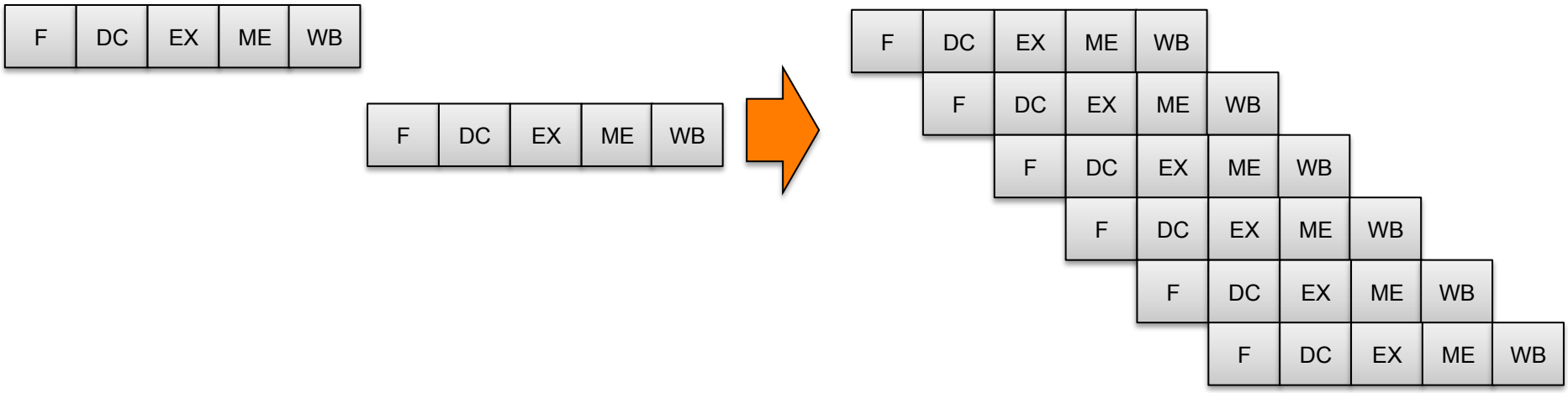
# Pipelining Review

Recall from last week's lecture:

We observed that a single instruction on a RISC machine goes through a number of stages as it is executed. Each step takes one clock cycle and uses a different piece of hardware: The instruction is shifted through the CPU much like a part on a factory assembly line

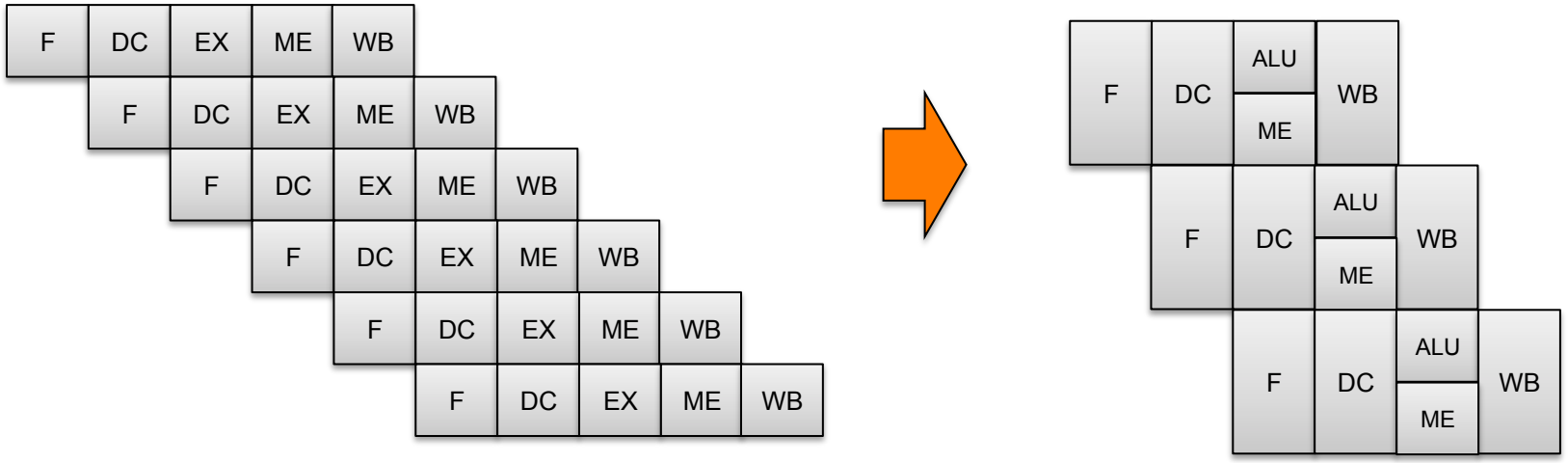
# Pipelining Review

Then, because each step takes different hardware, multiple instructions may be overlapped to improve throughput. This is called pipelining, each instruction takes the same amount of time to run however there are more being executed at any given time so throughput is improved



# Pipelining Review

We further noted that many different instructions have different requirements at the execute phase. For example, the instruction set may prevent any single instruction both using the ALU and having a memory access (like many RISC machines). By doubling the "width" of the other stages (i.e. allowing them to operate on two instructions at once), instructions may be executed completely in parallel with minimal duplication of hardware. This is an example of Superscalar design.



# Pipelining Review

Superscalar design can be extended further by noting that not all types of instructions occur with the same frequency. For example, integer operations are much more common the floating point, multiplication is more common than division etc. The ARM Cortex A8 for instance has three parallel execute blocks:

1. ALU/Multiply
2. ALU
3. Load/Store (memory)

In this particular example, the Cortex A8 can only issue two instructions per clock, like the system on the previous slide, so only two out of the three execute blocks are in operation at any given time.

# SIMD

In this limited example, two additions (say) can be done at a time but it requires two instructions.

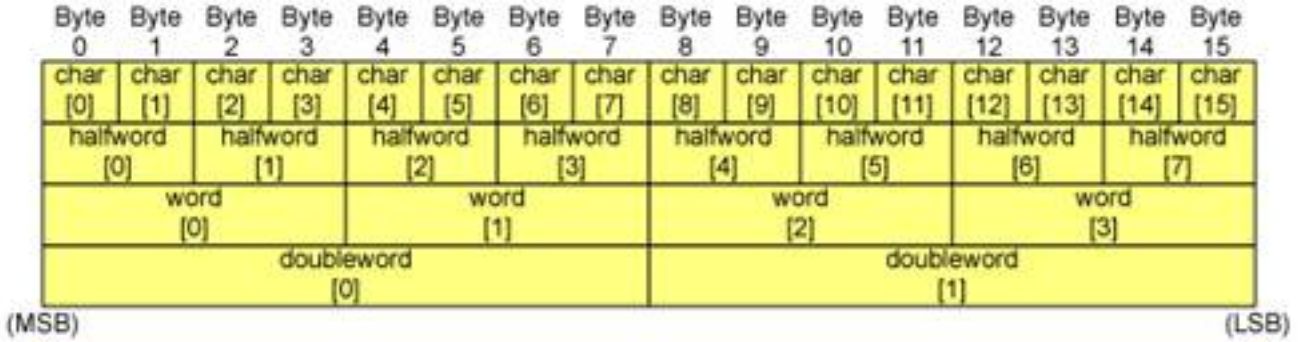
Pushing further, what if several registers could be loaded with inputs and a single instruction issued to perform an operation on all of them at once? This is called Single Instruction Multiple Data (SIMD) or Vector Execution (as opposed to [super]scalar execution). Most modern processors can perform some SIMD operations.

It is possible to do SIMD within a slightly modified ALU: For example, a 32-bit ALU may be able to be programmed to do four independent 8-bit operations at once, rather than a single 32-bit one. More commonly though, the processor is augmented with special “vector” registers that are much bigger than the usual.

# SIMD

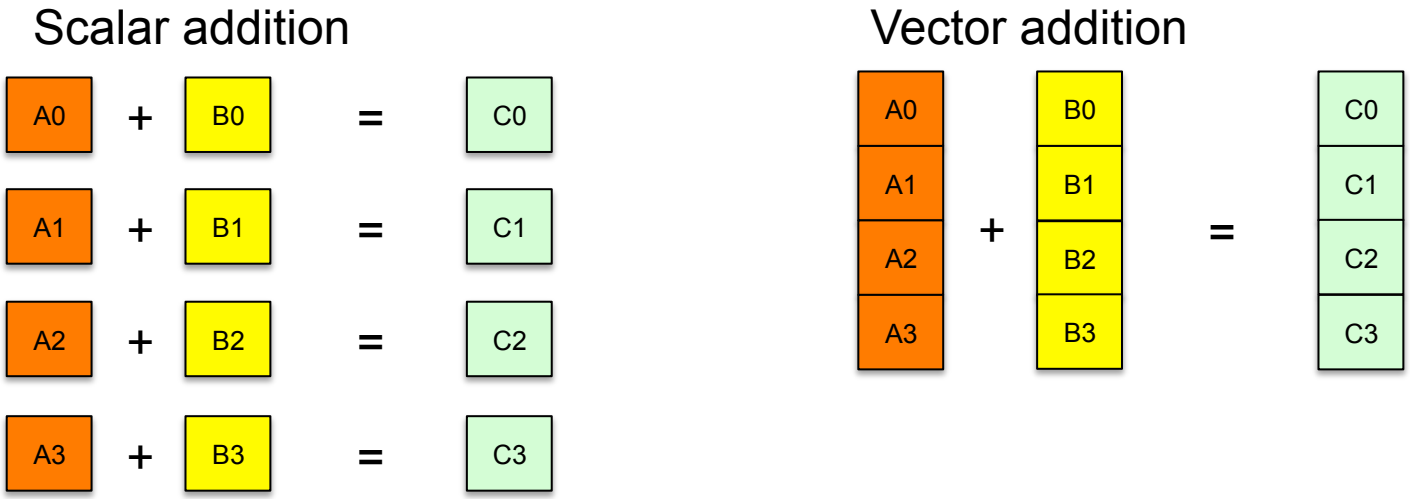
These vector regs are typically 128-bits long. As such, the a SIMD processor can typically execute 16 8-bit operations at once, down to two 64-bit operations.

Bigger vectors are of course possible however memory bandwidth becomes an increasingly large issue. No good being able to execute quickly if there's no data.



# SIMD

Once the vectors are assembled, it takes a single instruction to perform the computation



# SIMD

In order for this to work, the data has to be presented correctly in the input vectors. If the data is not natively available in the correct format, it may be more expensive to do the transformation then use a vector operation than just using a standard sequence of scalar operations.

For example, a brightness computation for a pixel requires an implementation of the following equation:

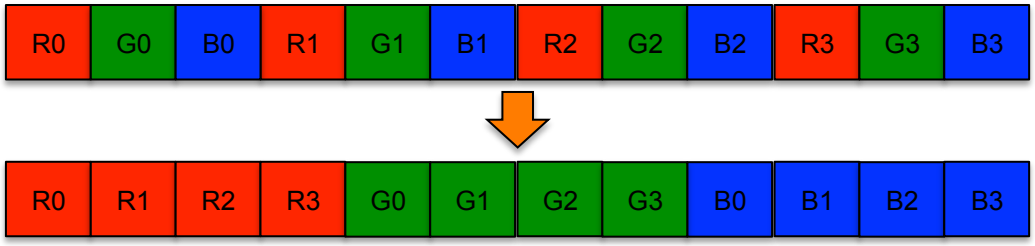
$$Y = R * 0.29891 + G * 0.58661 + B * 0.11448$$

This reflects the fact that our eyes are much more sensitive to green than the other colours.

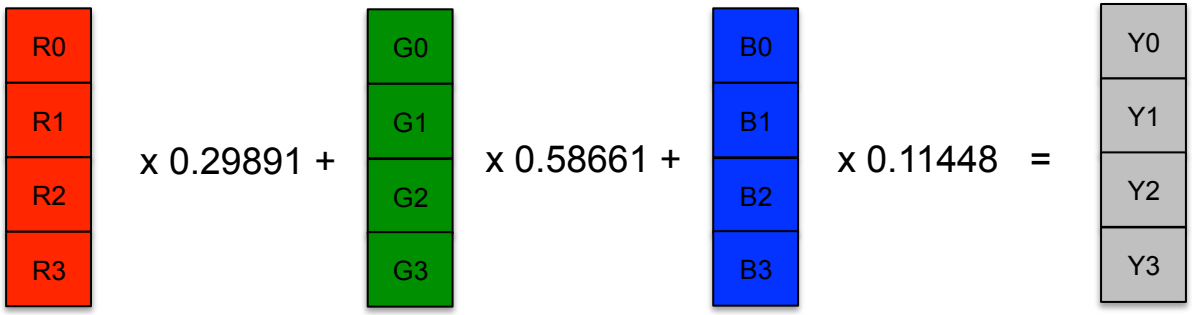
# SIMD

The problem is that pixel data is usually arranged as RGBRGBRGBRGB where as a vector operation would require it to be arranged as RRRRGGGGBBBB. This is called element permutation and is common enough that it is often vector accelerated as well.

Permute




Multiply and Accumulate



# SIMD

An example:

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[i]
```

Scalar Execution

```
LOOP:  Load B[i]
       Load C[i]
       Add
       Store A[i]
       Increment i
       Branch if i<N
```

Vector Execution, length L

```
LOOP:  LoadVect.L B[i]
       LoadVect.L C[i]
       AddVect.L
       StoreVect.L A[i]
       Add L to i
       Branch if i<N
```

Same number of instructions but by doing more at once, you reduce the number of iterations

# Stream Processing

Vector operations can be thought of as extending data operations "sideways", allowing more things to be done with a single instruction. What about extending the operations "lengthways"; allowing a single instruction to describe multiple operations on the same data? This is called Stream Processing.

A simple example is the Multiply and Accumulate (MAC) operation

```
x := x + a*b
```

This is used in computation of the dot product, matrix multiplication, evaluating polynomials and can accelerate some algorithms for doing division or finding a square root.

# Stream Processing

In Stream Programming, each operation to be perform is called a kernel. The sequence of kernels attached together is called the stream. Kernels may also be used to merge streams.

The following example is for a stereo imaging problem:

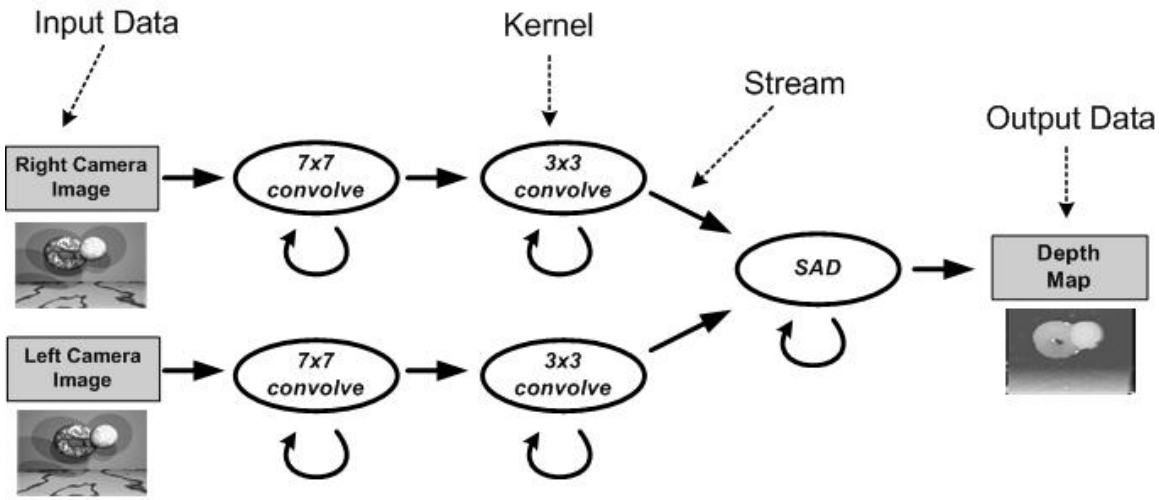



Image from Imagine Project, Stanford

# Stream Processing

Stream Processing has had limited success commercially, though it forms a key component of the Graphics Processing Units we will see later in the lecture. One interesting stream processor is the Stanford Imagine Storm-1 Processor.

It has eight ALU Clusters, each of which can implement several simple kernels in a stream. The clusters can further be chained together to form longer streams.

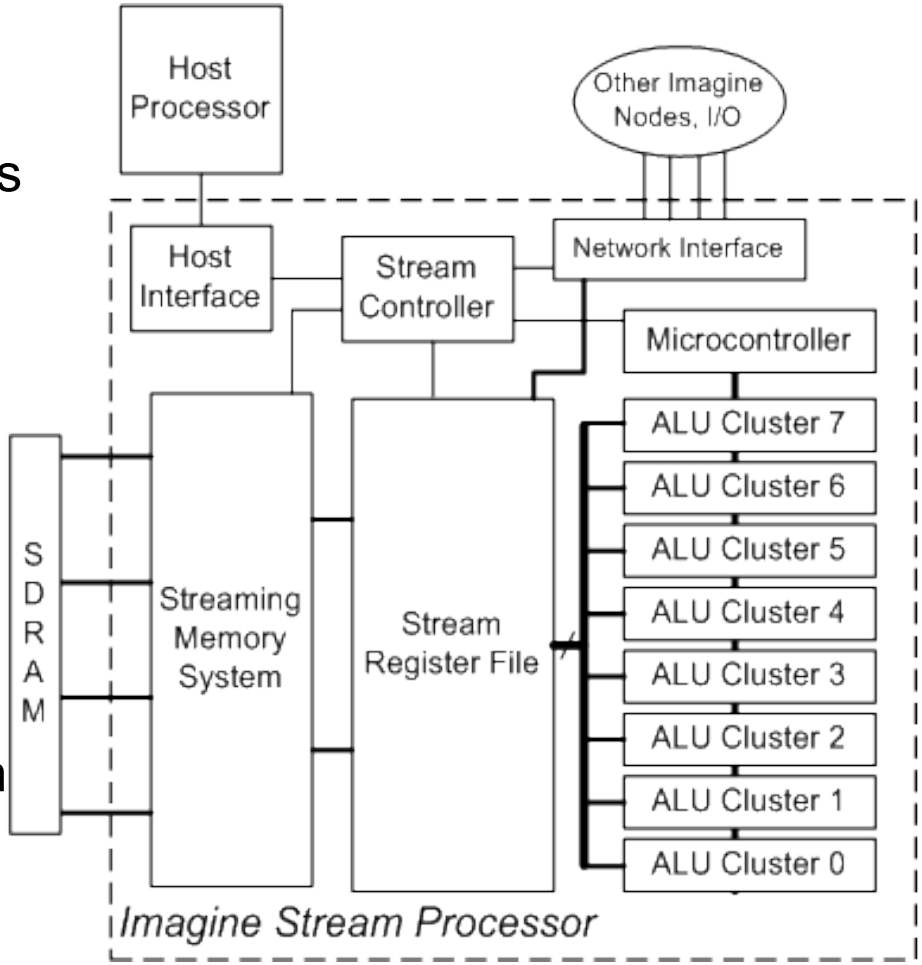



Image from Imagine Project, Stanford

# Stream Processing

These ALU Clusters are the base unit of programming in the Storm-1 architecture. They contain six ALUs, some control and storage elements.

The ALU Clusters are programmed and fed by a separate microcontroller. This also sets up the routing between the cluster and the Streaming Register File (SRF), which contains all input and output data

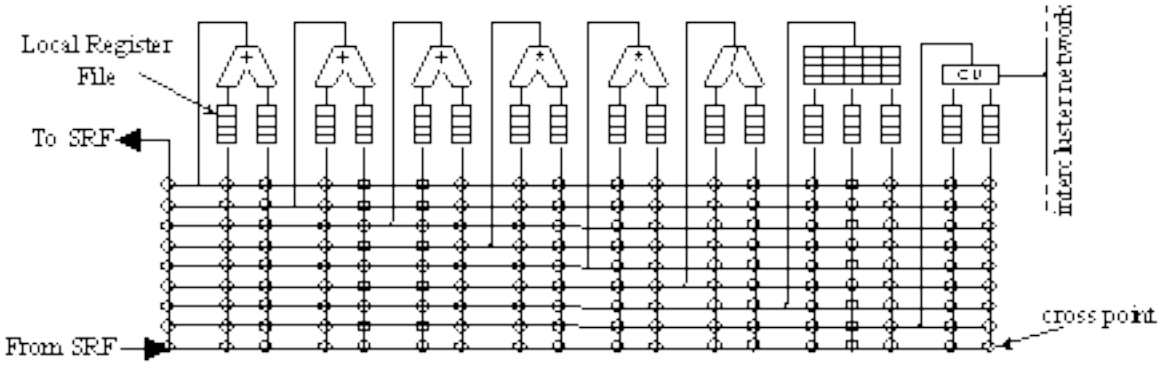
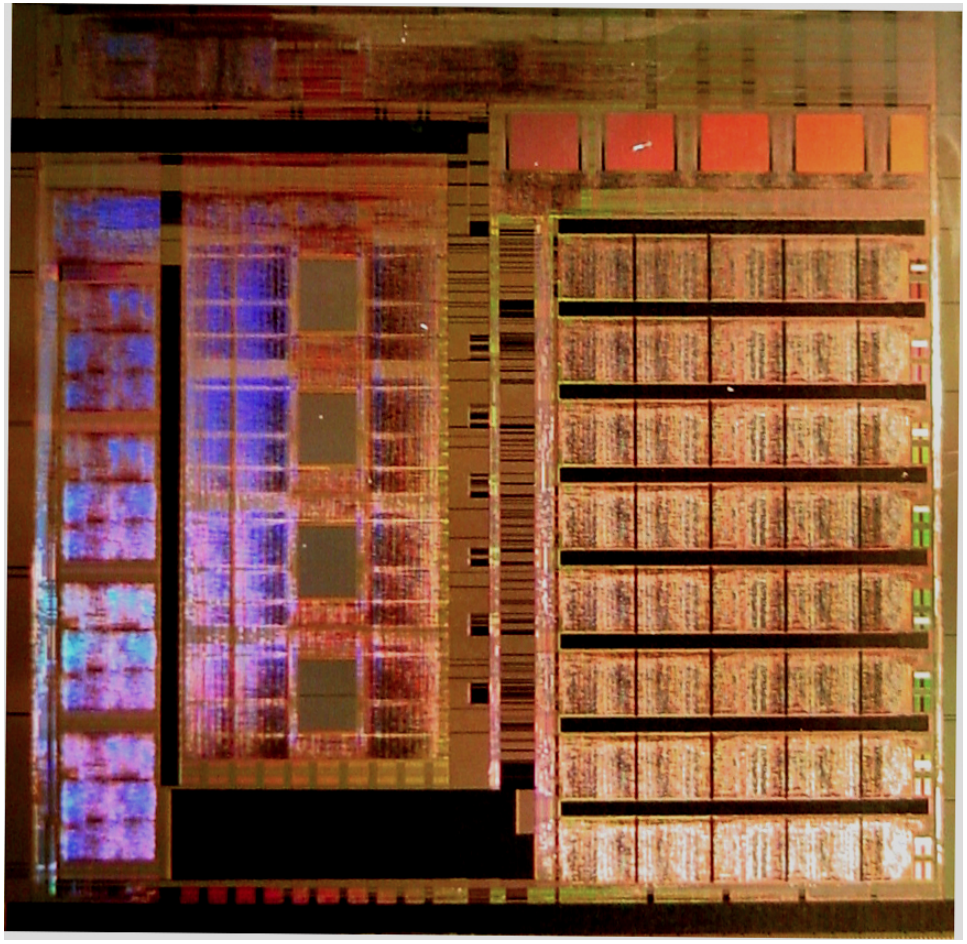



Image from Imagine Project, Stanford

# Stream Processing

Stream processors have large memory bandwidth requirements. That's the down side of being able to process data quickly!

In the Imagine Storm-1 processor to the right, you can clearly see the 8 ALU Clusters on the right, everything on the left is memory interfacing and registers.



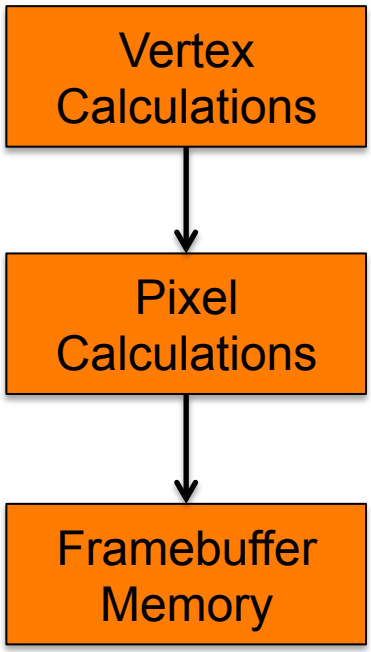
# GPU

We have seen SIMD architectures that allow multiple pieces of data to be processed simply with one instruction. We have then seen Stream processors that allow one piece of data to be processed with several instructions in turn. Putting this all together, we get a processor that can perform multiple instructions on multiple data at once.

The canonical example of where this is useful is in graphics processing: The operations will be rendering, shading, texturing; across lots of vertex, triangle, image or pixel data.

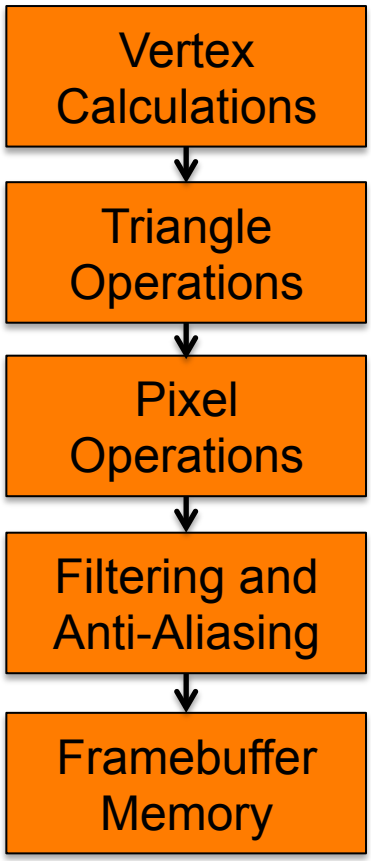
# GPU

Originally GPUs didn't have much to do: Turn vertices in to visible triangles, turn triangles in to pixels. Each of these used its own specialised hardware




# GPU

As things evolved, more hardware was added to put textures on triangles, filtering and anti-aliasing. Pixel operations were extended to simplify reflections, shadows.




# GPU

It was observed that with this complex sequence of operations, it wasn't likely that all the hardware modules would be used equally. As such, hardware was often sitting idle, wasted.

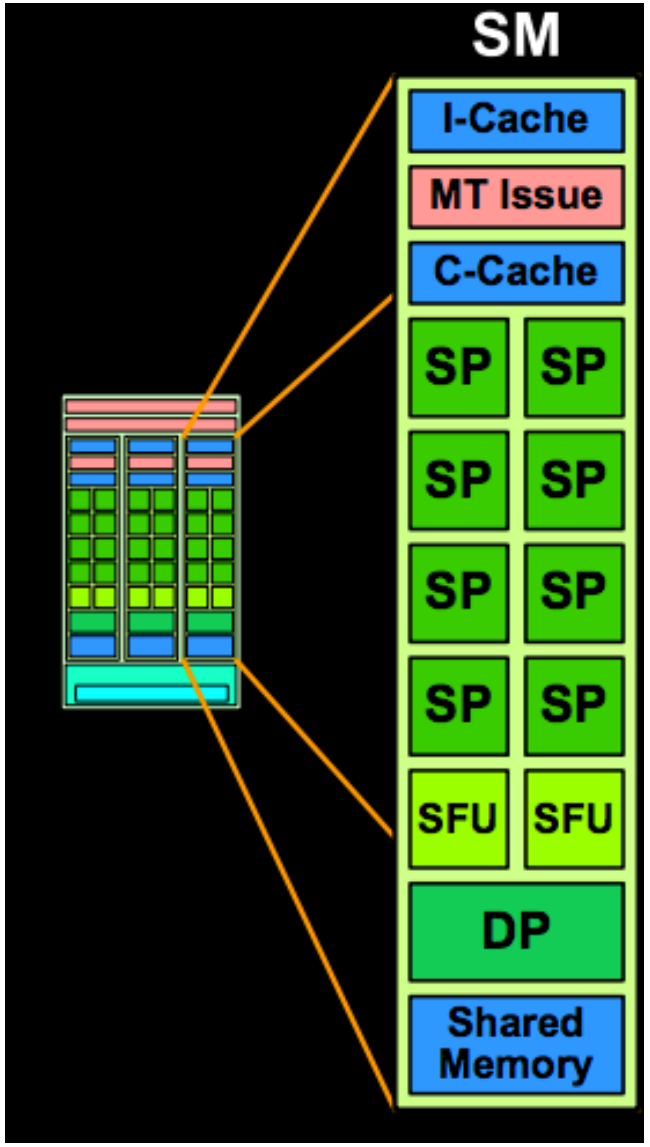
Geometry-heavy

Pixel-heavy

# GPU

Specialised hardware has been replaced with more generic hardware, able to be reconfigured to perform whatever types of graphical operations are required.

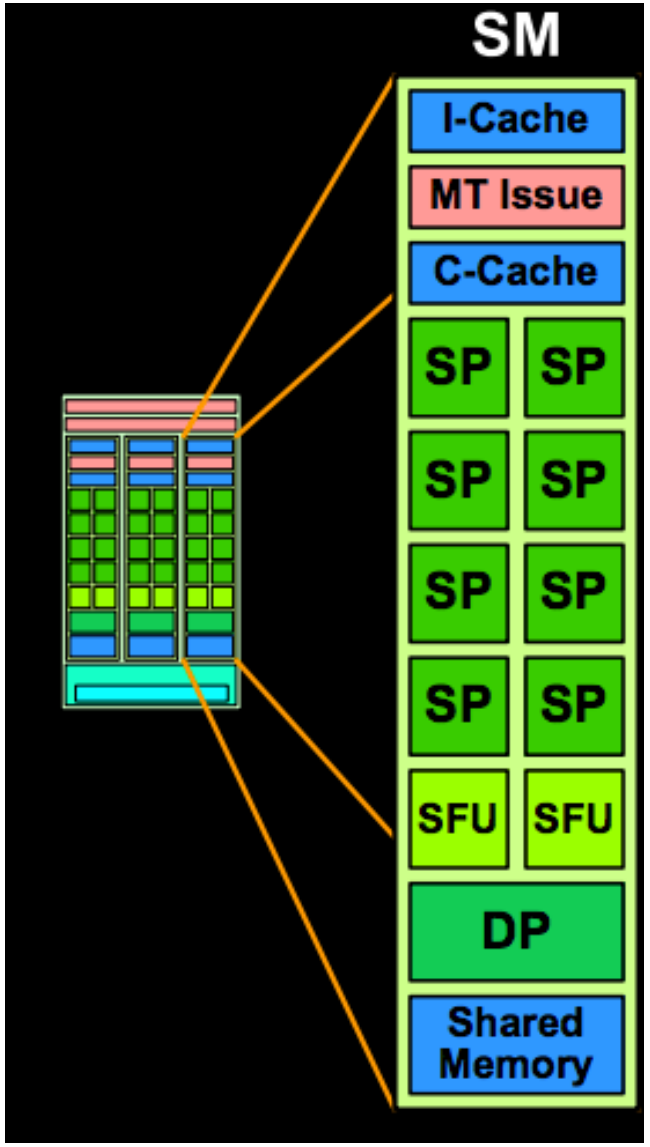
The following example is from a modern NVidia card and uses their terminology, but the concepts are equally applicable to other brands.



# GPU

A modern GPU contains several Stream Multiprocessors (SM), each containing several Stream Processors (SP). Each SP contains an ALU and Floating Point unit.

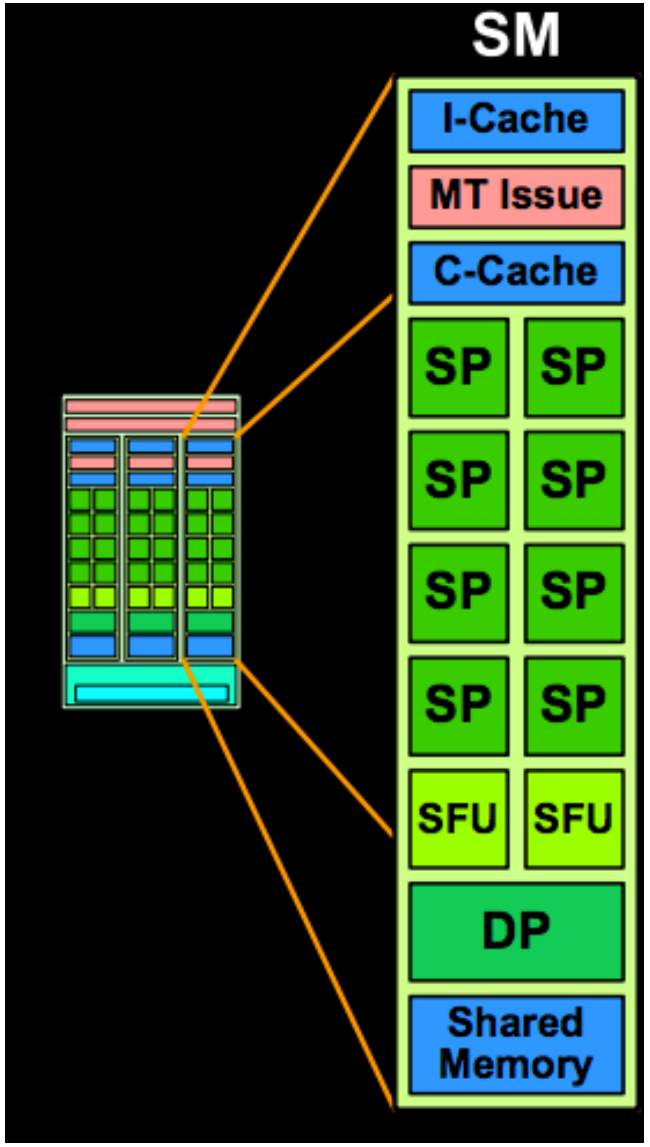
In addition, each SM contains some Special Function Units (SFU) that perform in hardware things that are inefficient in general units, e.g. trig functions, logarithms etc.



# GPU

Each of these SMs loosely corresponds to a Stream Processor as defined previously: Multiple operations run in sequence over the same data. Modern SMs are more powerful than the stream concept before as they aren't strictly linear, they may throw away data chunks halfway through, perform conditional operations, split and merge streams internally etc.

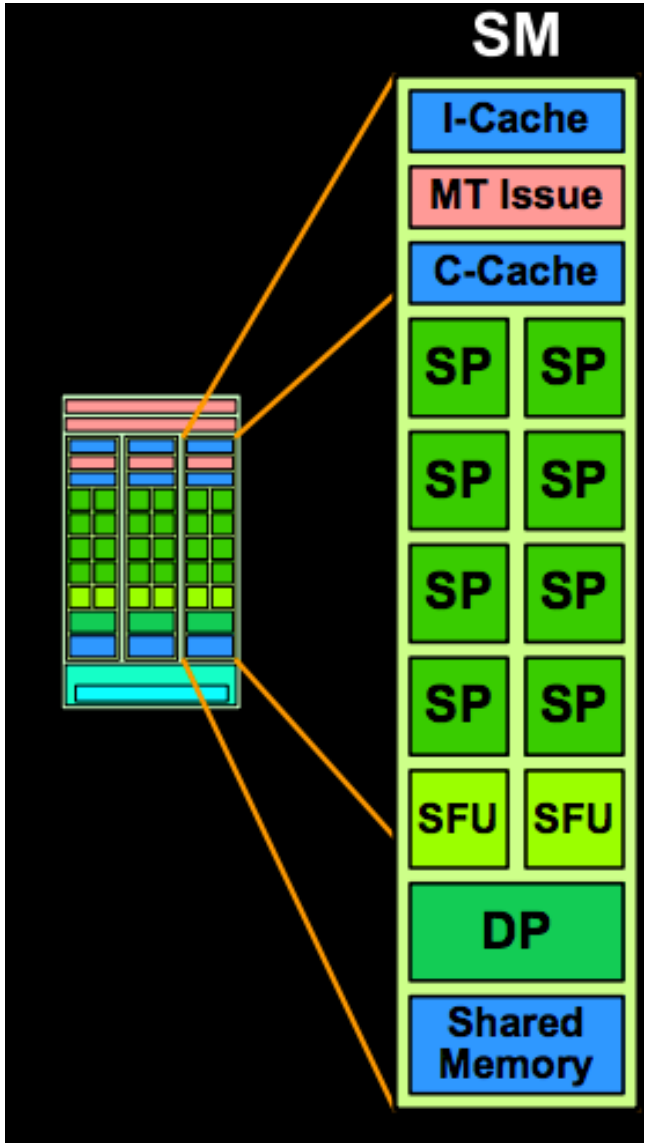
In terms of graphical operations, these may be things like only rendering pixels that are actually visible



# GPU

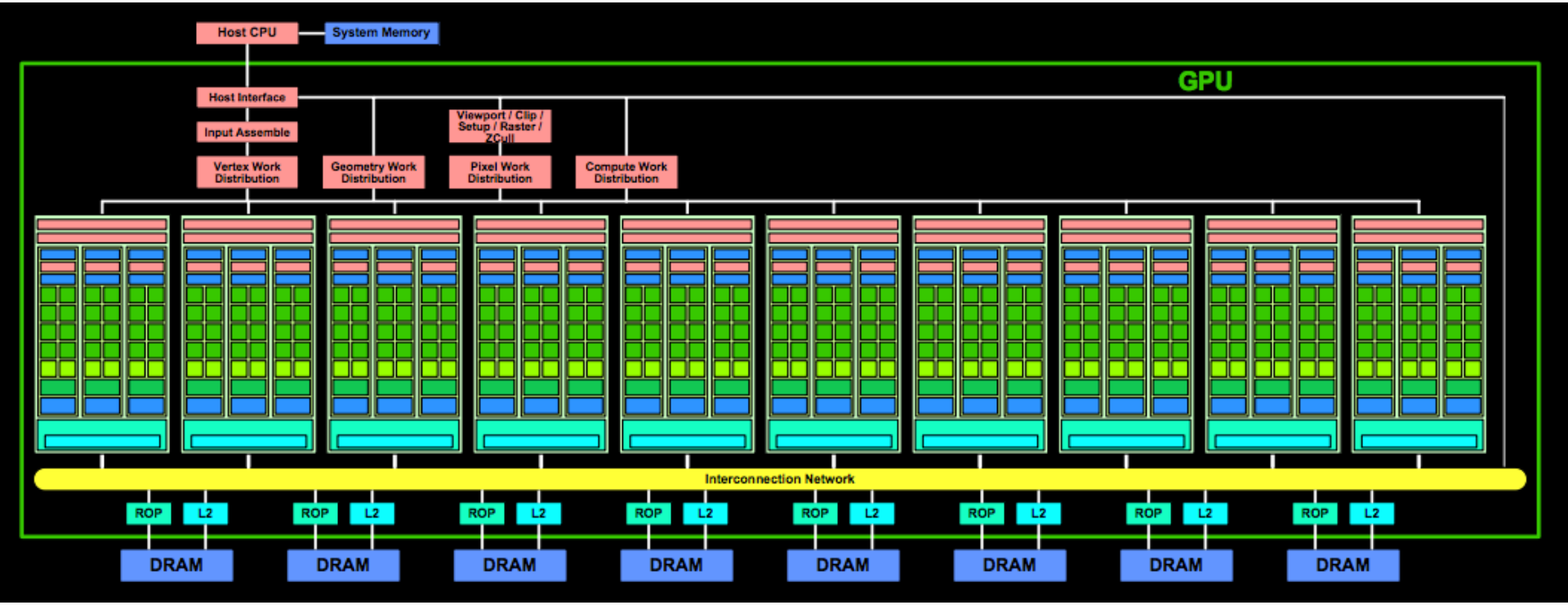
Each SM includes Multithreaded Issue logic. This further adds to the flexibility of an SM by allowing several different sets of instructions to be run on the same SM, just not at the same time.

This is completely analogous to being able to run several different programs on your PC at once, despite the fact that the CPU may only be able to execute one instruction at a time.



# GPU

A GPU has multiple SMs, each of which can run largely independently of each other. In a game scenario, some SMs may be dedicated to rendering while others are doing physics calculations.



# GPU

With all these SMs and their component SPs and MT Issue logic, it is apparent that a modern GPU may be able to run several thousand sequences of operations (threads) at once. Each thread is similar to what you may find on a CPU, a series of instructions that guide the processor elements to loading data, storing data and performing data transformations.

You may remember from the previous lecture that in some circumstances a processor can execute up to two threads at once (Superscalar Execution), but even in that case full utilization was unlikely as the threads would have interdependencies that would cause a pipeline to stall.

# GPU

How can a GPU support high utilization with 10,000 simultaneous threads? More than that, how can a programmer write 10,000 threads?

Problems that can be solved on a GPU must obey strict properties:

1. All the threads must be independent of each other so there are no data or synchronization issues: They fit in to Streams
2. The threads must share the same instructions but will operate on different data: SIMD. This means the programmer only needs to write the thread once, even if its run 10,000 times

In practice, the programmer writes a small number of threads, each of which is run a few thousand times; you can run more than one type of thread on a GPU at any given time.

# GPU

These threads are also called fragments. Here's a shader fragment that computes "diffuse reflectance".

It gets the colour of the texture at the location then makes it brighter or darker based upon the relative angle of the surface and the light source

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= dot(lightDir, norm);
  return float4(kd, 1.0);
}
```

# GPU

Does the fragment comply with the two simple rules laid down before?

Is it SIMD? That is, will the same instructions apply cleanly to every possible input?

Yes, by construction the fragment can be run across all inputs at once. Further, internally the fragment uses vector operations.

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= dot(lightDir, norm);
  return float4(kd, 1.0);
}
```

# GPU

Does the fragment comply with the two simple rules laid down before?

Is each instance independent?

Yes, while “mySamp”, “myTex” and “lightDir” are all shared, they are all inputs only; they aren’t changed by any fragment instance.

```
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;

float4 diffuseShader(float3 norm, float2 uv)
{
  float3 kd;
  kd = myTex.Sample(mySamp, uv);
  kd *= dot(lightDir, norm);
  return float4(kd, 1.0);
}
```

# GPU

GPUs have been presented as devices that have evolved in to highly parallel, general purpose maths engines. That mathematical ability is generally used for graphics and gaming, but it is clear that there is no particular restriction to that feature set.

Under what circumstances would one implement their particular algorithm on a GPU?

# GPU

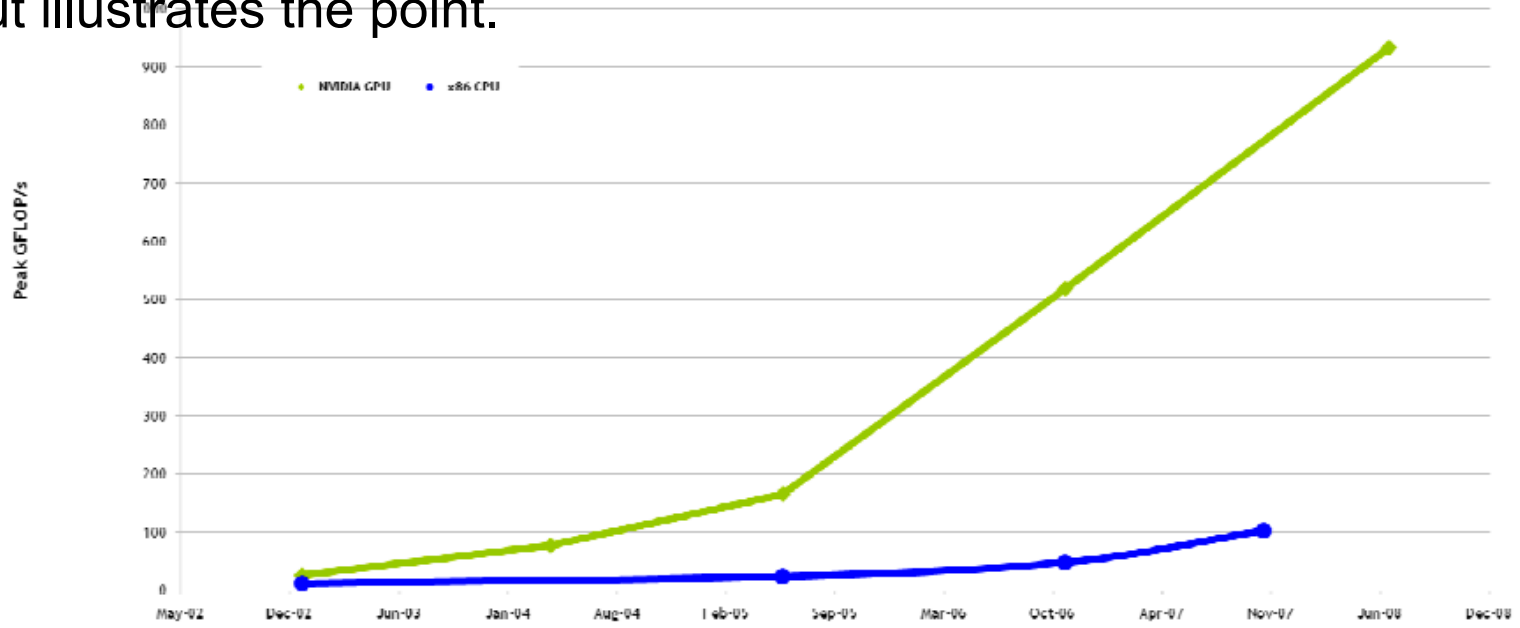
The problem to follow the two rules laid out previously. That is, it must be able to be decomposed in to relatively few sets of instructions, each of which can be executed unchanged across multiple data. Further, each of these sets of instructions must be able to run independently of each other when run on different data. This is a fairly restrictive requirement and may make the decision for you.

Then the decision comes down to the data set. The data set must be large enough that the speed up of the GPU outweighs the cost of initializing the GPU and transferring the data back and forward.

# GPU

The throughput of GPUs is now at least an order of magnitude above that of a CPU, so that break-even point is getting ever-easier to reach.

This graph is taken from some relatively old NVidia marketing material but illustrates the point.



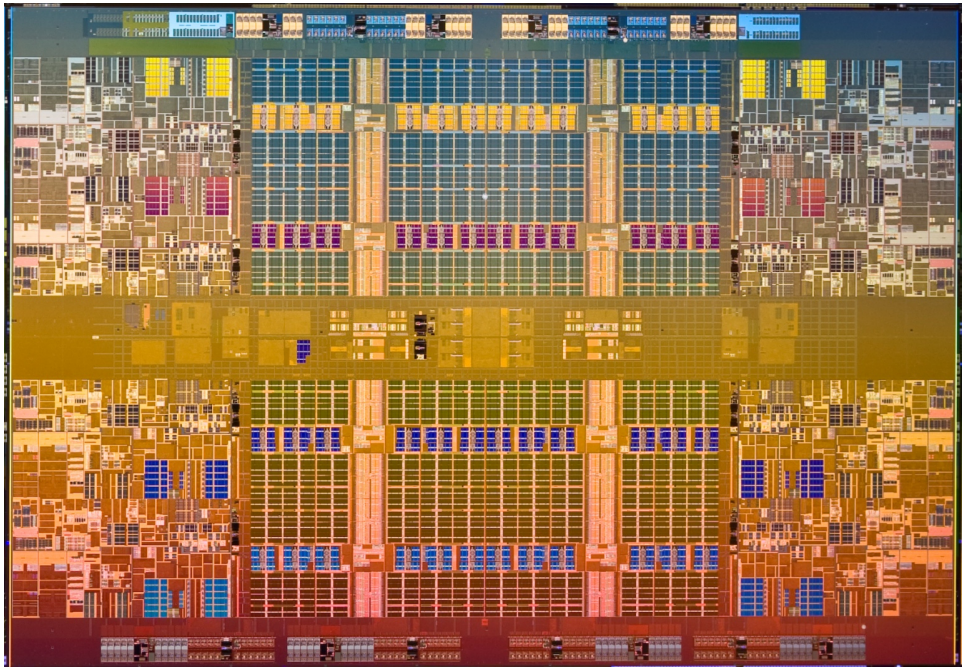
# GPU

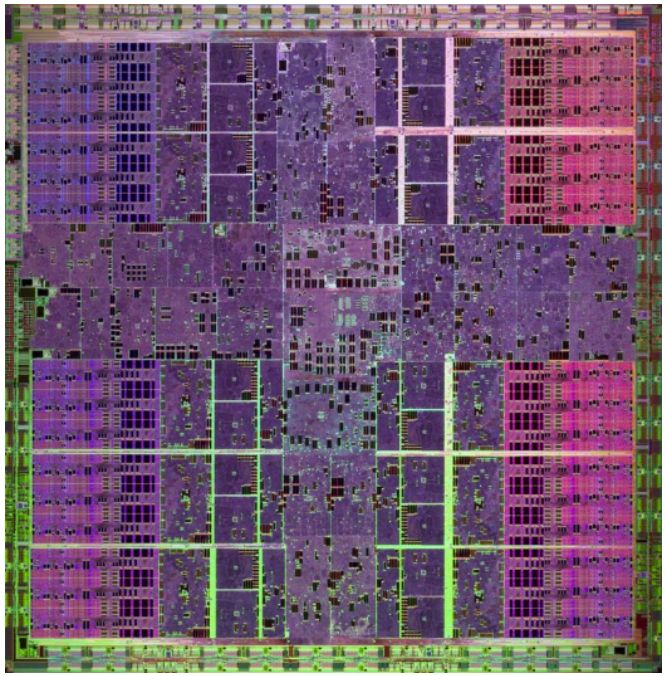
A question arises: If packing hundreds or thousands of floating point units on to a chip is a good idea, why isn't it done as part of normal CPU pipelines? First, remember why we need Cache on a CPU: To reduce latency on memory accesses so the CPU doesn't sit idle waiting for data to turn up. On a CPU, we can say that memory access latency is masked with caches. This doesn't work well for large data sets that don't fit in cache (see Tutorial Question 1).

In a GPU data sets are so large that cache would be largely useless, so it is left off. Instead, latency is masked by threading. If a GPU thread is waiting for memory then a different thread is scheduled. A GPU may have 10's of thousands of threads to choose from, so one can usually run!

# GPU

We can then answer our question about packing more execution ability on to a CPU: It won't fit along with a Cache!





# Tutorial Question 2

Consider the pipelined execution of the following operations:

```
R4 = R3 + R2
R7 = R6 | R5
R8 = R7 - R4
```

# Tutorial Question 2

1. How many clock cycles would it take to complete these operations on a canonical RISC machine with no operand forwarding?
2. How many clock cycles would it take if the EX stage output could be forwarded to its input?
3. How many clock cycles would it take if the EX and MEM stage outputs could be forwarded to the EX input?
4. Now assume that instead of R5, the second operand of the OR has to be fetched from L2 Cache with a 10 cycle latency. Draw a diagram similar to Lecture 3, Slide 18 but with three operations in flight, showing the duration of the stall and how long the execution would take overall now