Hardware for Machine Learning:

Computations on graphics processors

Ingemar Ragnemalm

Information Coding, ISY

This lecture:

GPU evolution and GPU architecture

How to write simple CUDA programs

How to port from the CPU

How to optimize

Alternatives: OpenCL, GLSL, Compute shaders

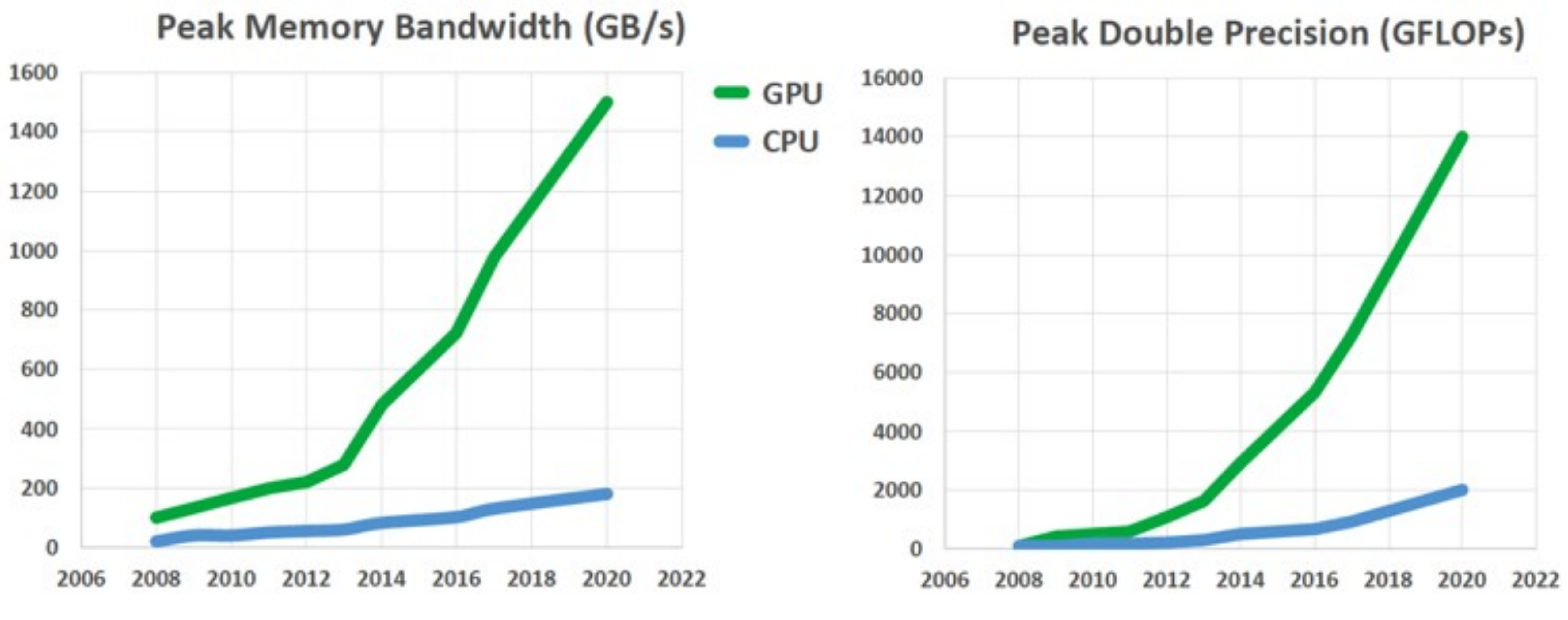
Lecture questions:

1. Why did the GPU evolve into a general purpose parallel processor?

2. What operations do tensor cores accelerate?

3. How can you limit global memory access in CUDA?

The GFLOPS race



GFLOPS in numbers:

	GPU	CPU
1995:	0.001	0.09
2005:	40	5.6
2011:	2488	91
2015:	7000	176
2016:	16380	400-700*
2017:	110000**	4000**

* Theoretical, 16 cores
** Claimed by NVidia, Titan V
*** Theoretical peak performance

Gets complicated here:
CUDA vs tensor cores

(Various sources)

How about economy: dollar per GFLOPS?

1961:	8.3 trillion
1984:	42 million
1997:	42000 (CPU cluster)
2000:	836-1300
2007:	52
2012:	0.73 (AMD 7970)
2013:	0.22 (PS4)
2015:	0.08 (Radeon R9 295)

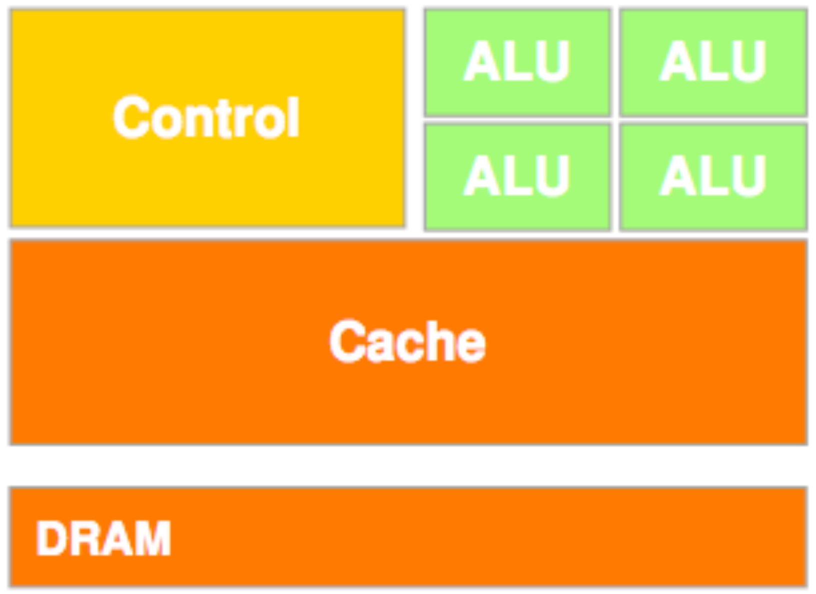
(Wikipedia)

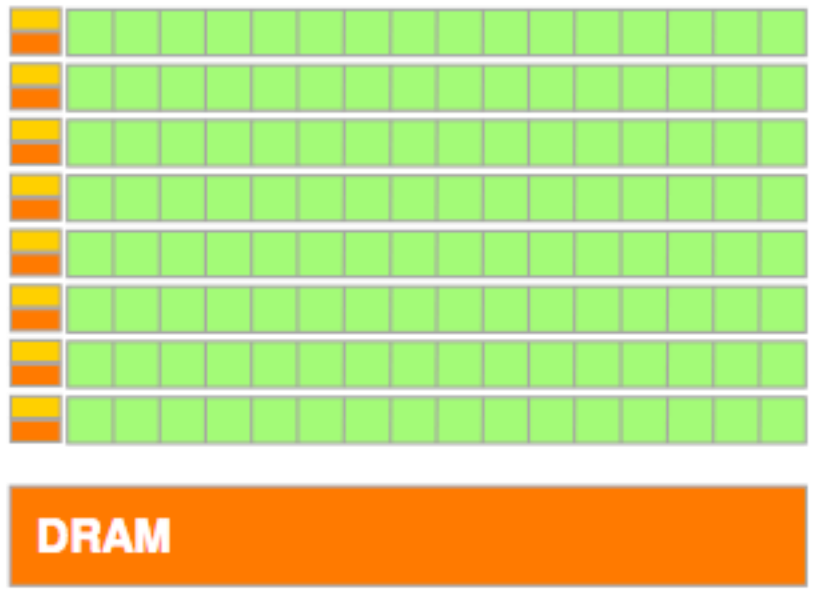
How is this possible?

Area use:

CPU

GPU





But in particular: SIMD architecture

SIMD

Single instruction, multiple data

Simplifies instruction handling. All cores get the same instruction.

Excellent for operations where one operation must be made on many data elements.

Is that so common? Yes!

Data best in stored arrays.

SIMT - Single Instruction, Multiple Thread

A variant of SIMD.

Parallelism expressed as threads.

A programming model, but also demands that the hardware can handle threads very fast.

Threads dependent - executed in a SIMD processor!

So, why does SIMT fit a graphics processor so well?

Data Oriented Programming

DOP optimizes for performance.

Data structures selected to fit the computations, instead of the programmer!

Optimize for the end user instead for the programmer!

Popular in the game industry - why not elsewhere?

Major past and current success stories:

Crypto currency

Bitcoins, Litecoins and others.

Deep learning

Learning systems based on very large neural networks.

Why did GPUs get so much performance?

Early problem with large amounts of data. (Complex geometry, millions of output pixels.)

Graphics pipeline designed for parallelism!

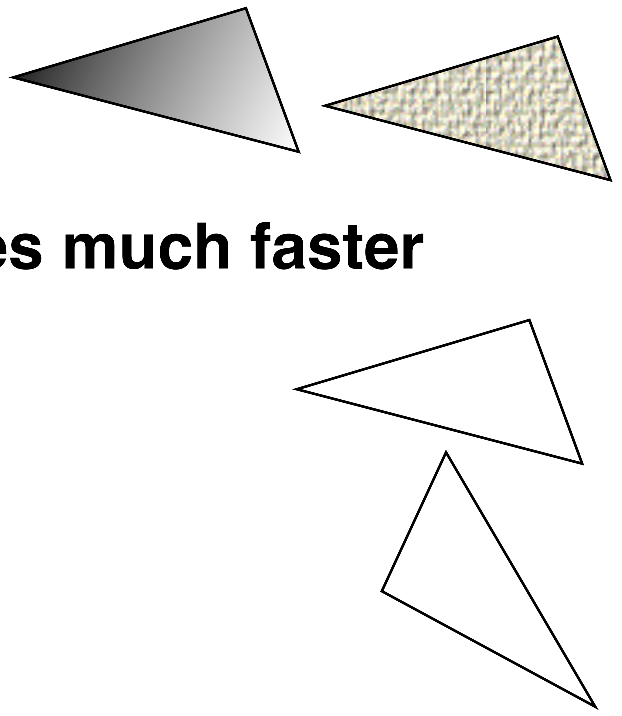
Hiding memory latency by parallelism

Volume. 3D graphics boards central component in game industry. Everybody wants one!

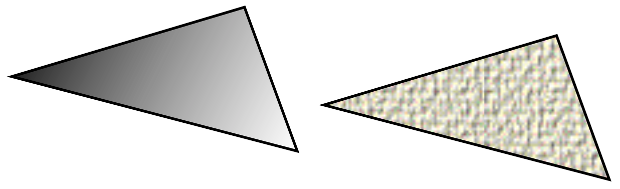
New games need new impressive features. Many important advancements started as game features.

Must process many pixels fast!

Early GPUs could draw textured, shaded triangles much faster than the CPU.

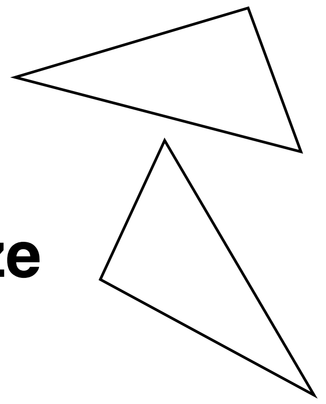


Must process many pixels fast!

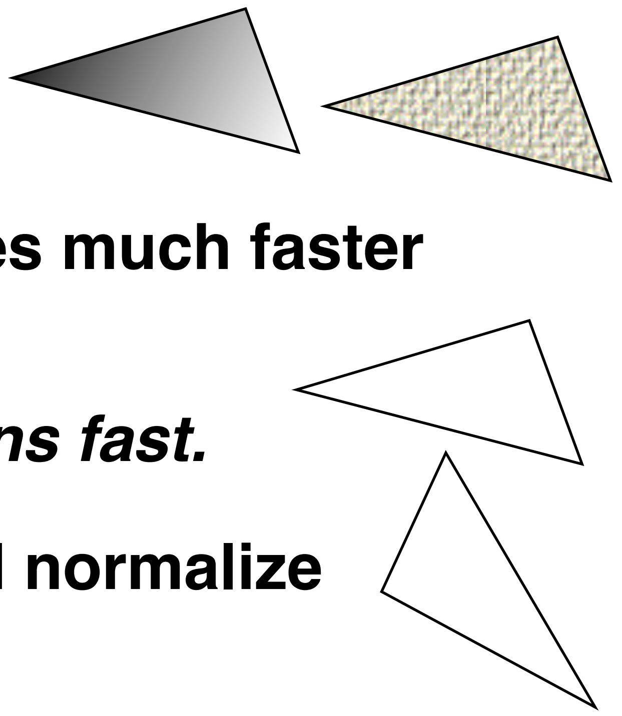


Early GPUs could draw textured, shaded triangles much faster than the CPU.

Must do matrix multiplication and divisions fast.



Next generation could transform vertices and normalize vectors.



Must process many pixels fast!

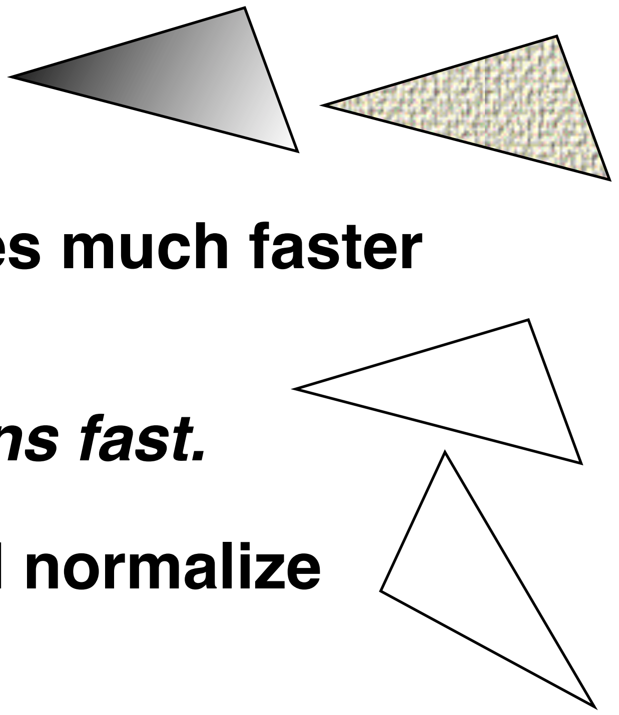
Early GPUs could draw textured, shaded triangles much faster than the CPU.

Must do matrix multiplication and divisions fast.

Next generation could transform vertices and normalize vectors.

Must have programmable parts.

This was added to make Phong shading and bump mapping.



Must process many pixels fast!

Early GPUs could draw textured, shaded triangles much faster than the CPU.

Must do matrix multiplication and divisions fast.

Next generation could transform vertices and normalize vectors.

Must have programmable parts.

This was added to make Phong shading and bump mapping.

Must work in floating-point!

This was for light effects, HDR.

A look at the GPU architecture

Over to the timeline, big changes:

Pre-G80: Separate vertex and fragment processors.

Hard-wired for graphics. Load balance problems.

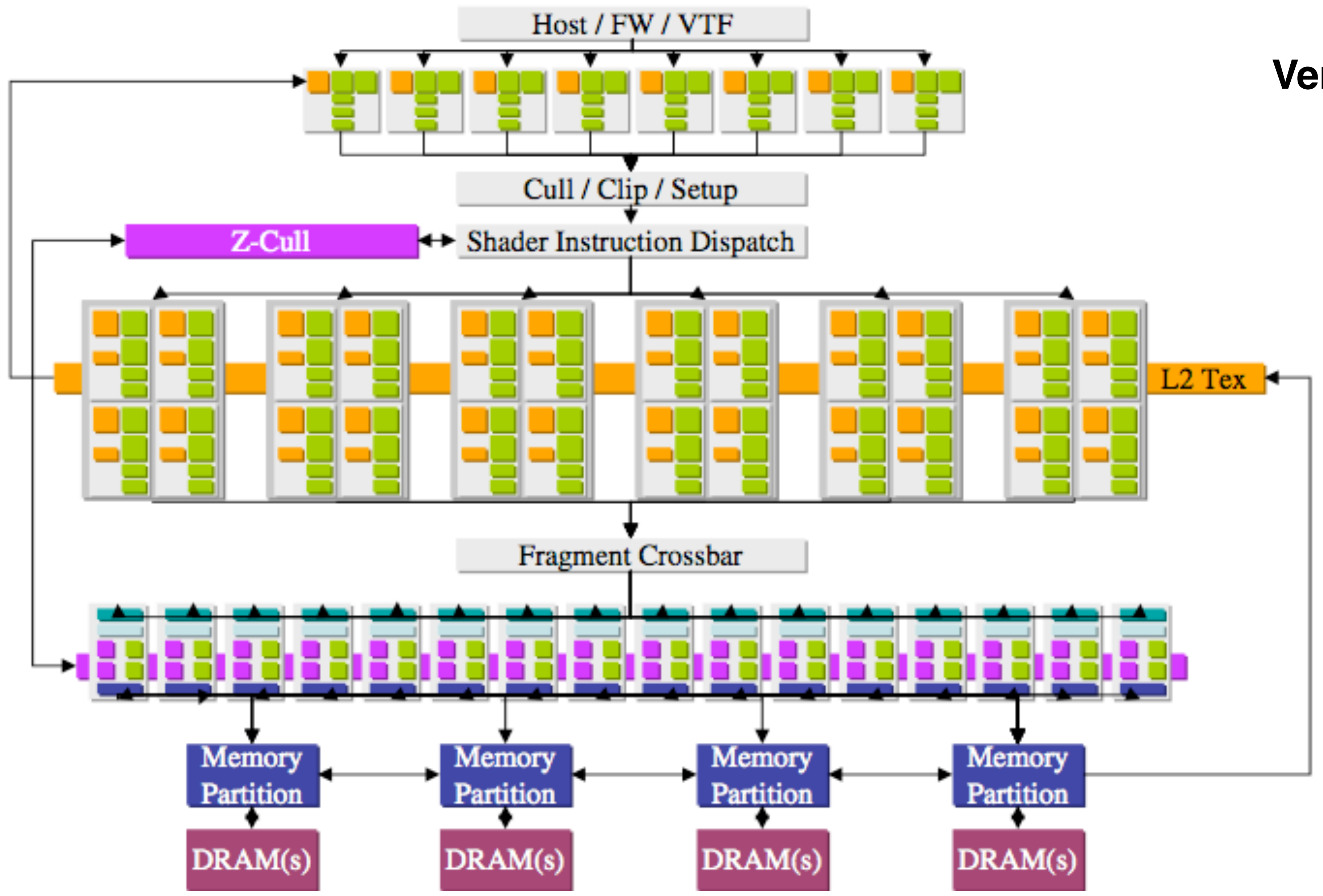
G80: Unified architecture. More suited for GPGPU. Higher performance due to better load balancing.

GT100: Much more double precision

TU102: Tensor & RT cores

(Similar track for AMD)

7800: High-end GPU before G80

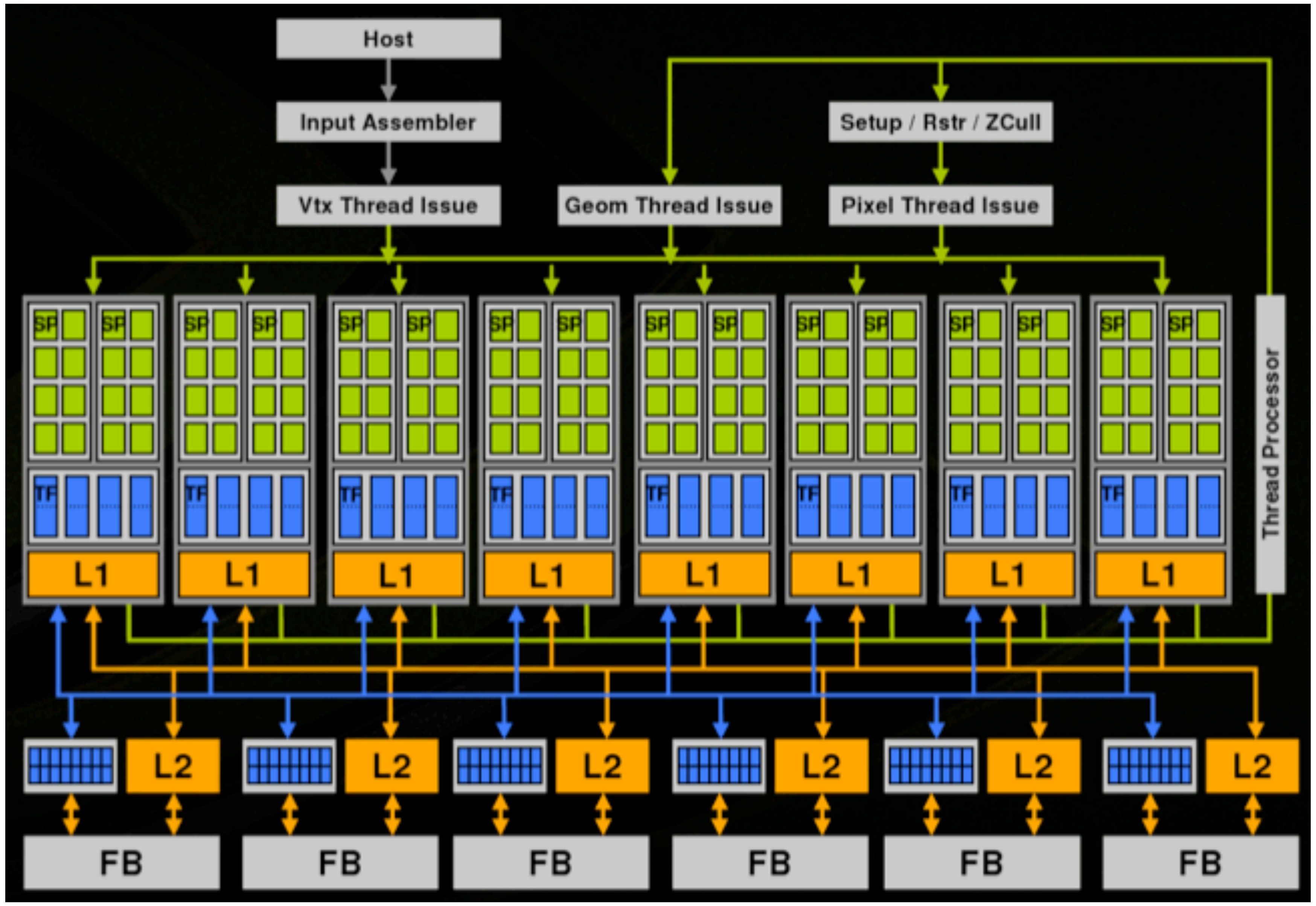


Vertex processors

Fragment processors

Framebuffer operations

G80

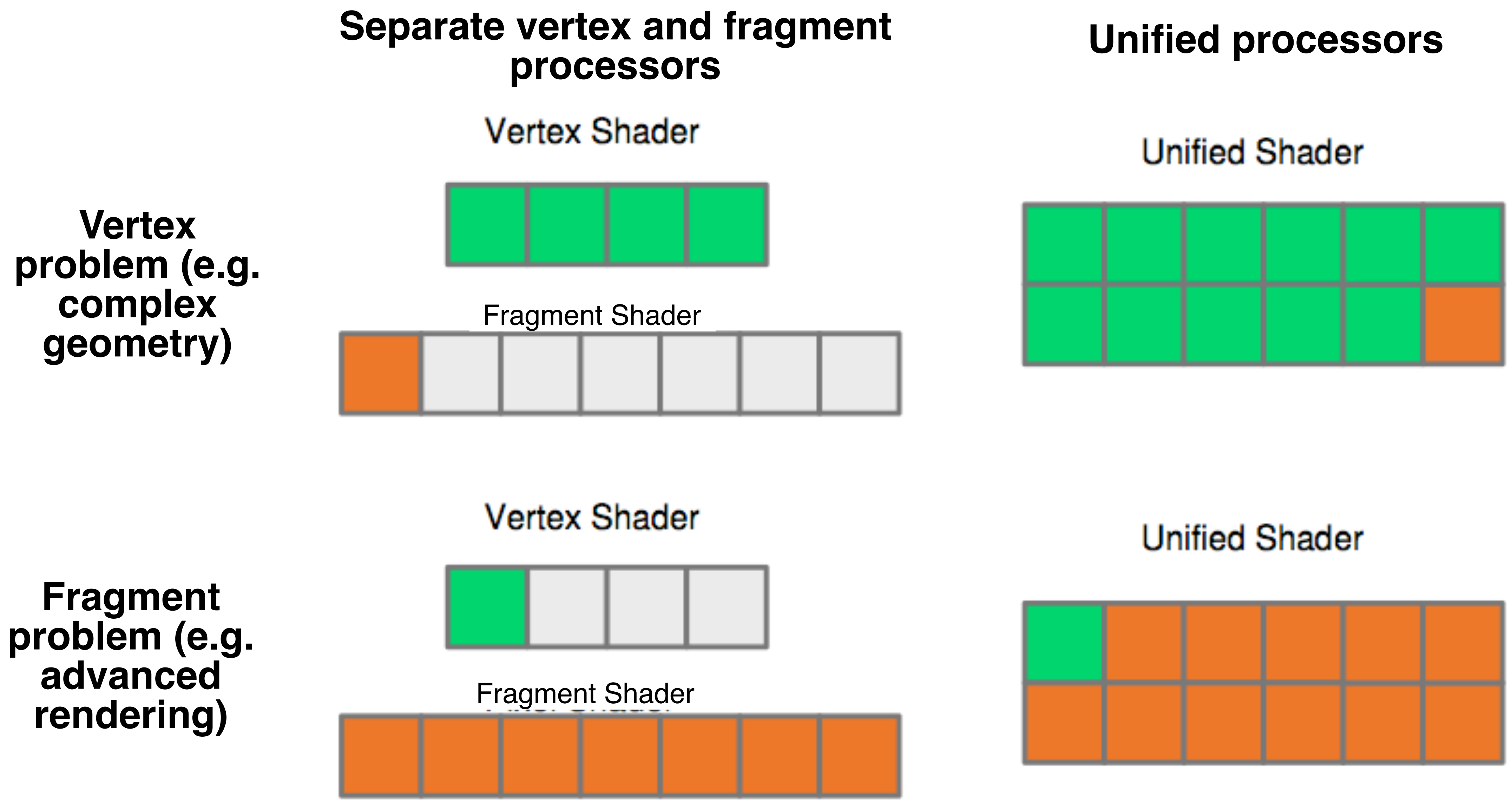


Hardware formerly between vertex and fragment processors

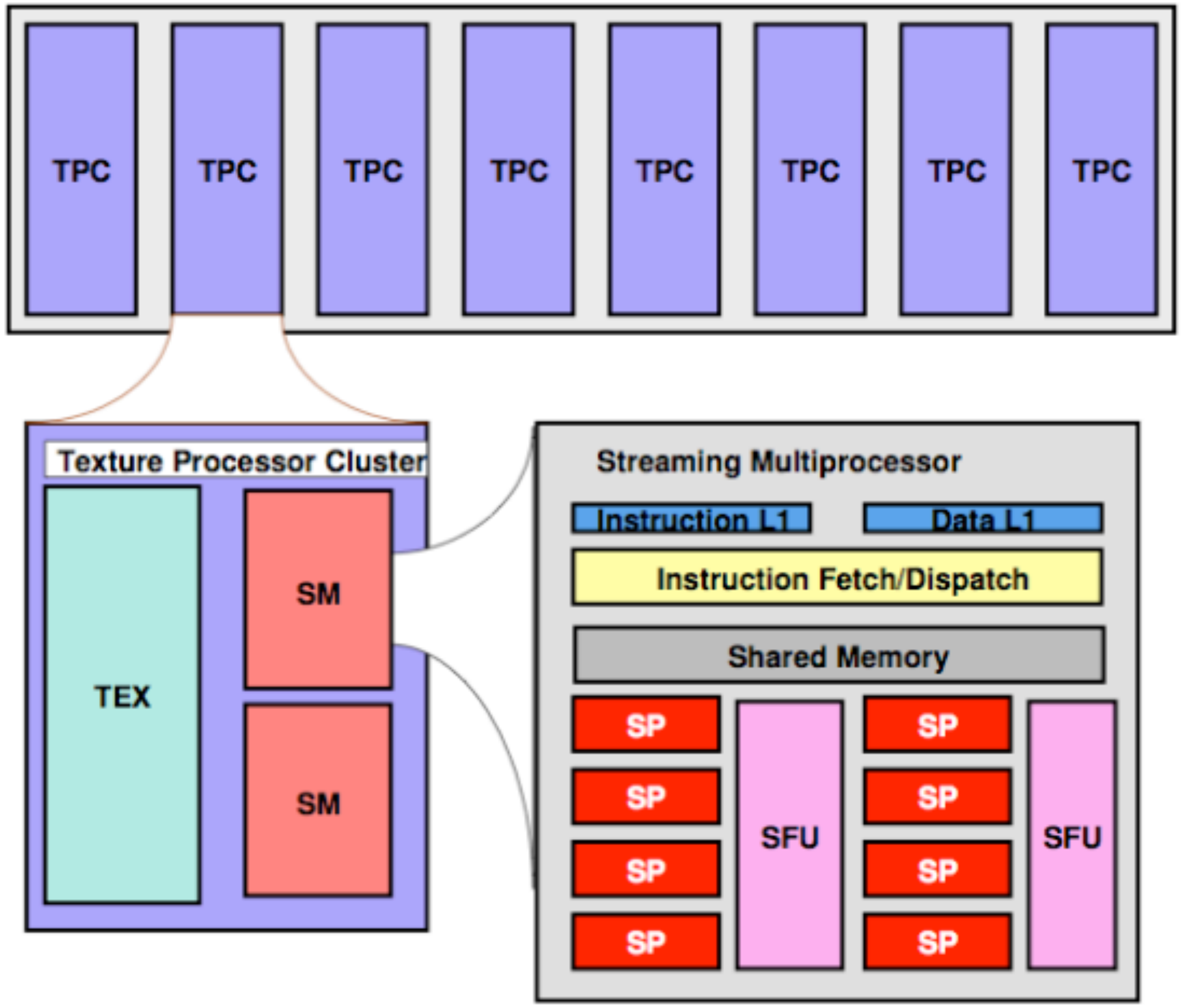
Unified processors!

Framebuffer operations

G80: A question of *load balance*!


Separate vertex and fragment processors
Unified processors
Vertex problem (e.g. complex geometry)
Vertex Shader
Fragment Shader
Unified Shader
Fragment problem (e.g. advanced rendering)
Vertex Shader
Fragment Shader
Unified Shader

G80 processor hierarchy

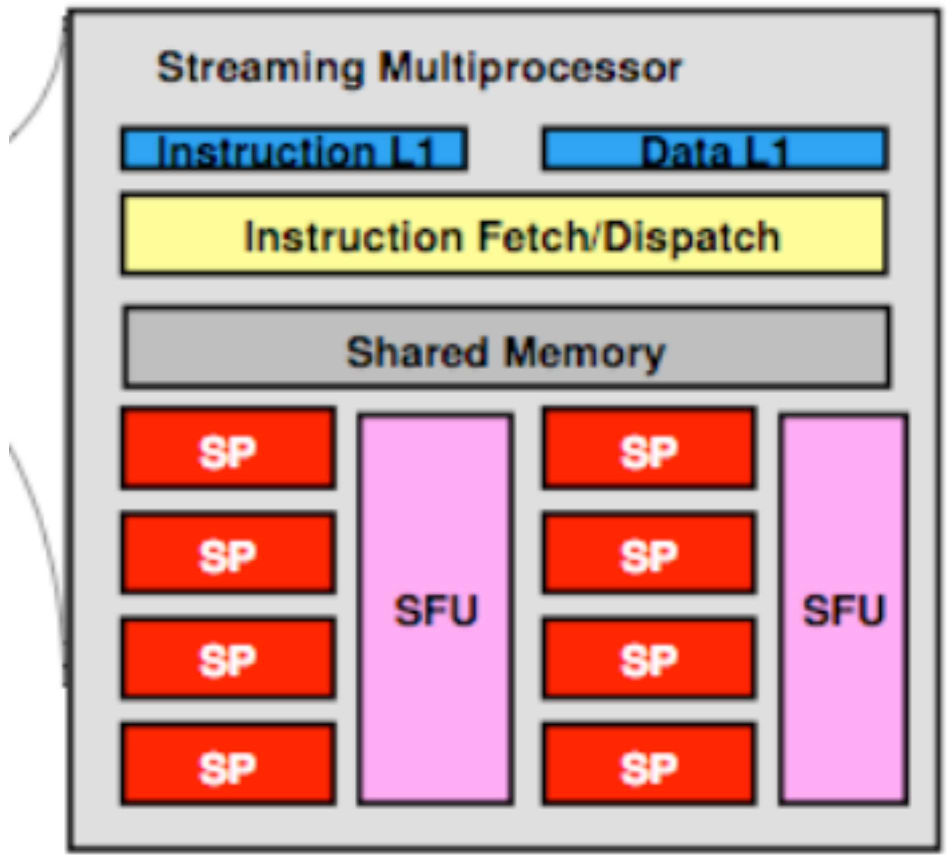


8 top-level groups of TPCs

SM = Streaming Multiprocessor

SM is a group of 8 SIMD cores

The vital part: The SM



SM: 8 cores

but also

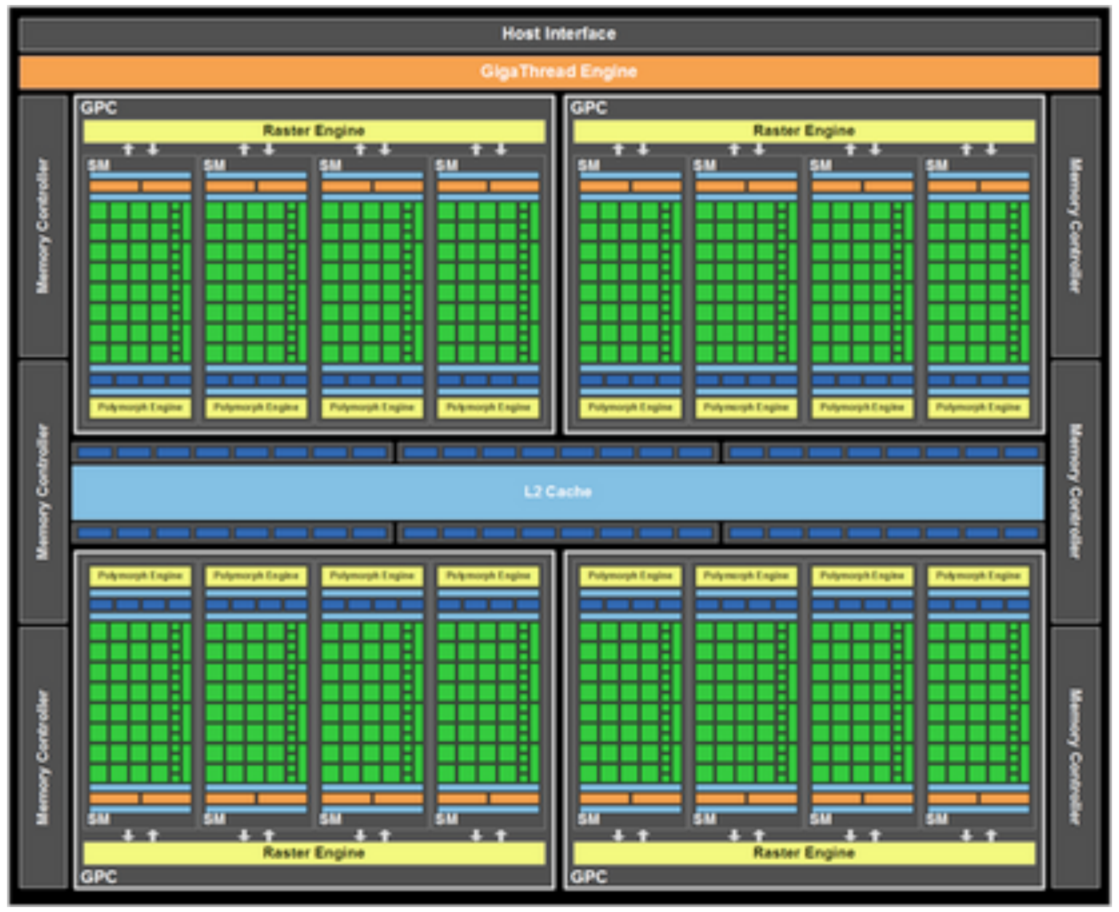
SFU: Special functions unit

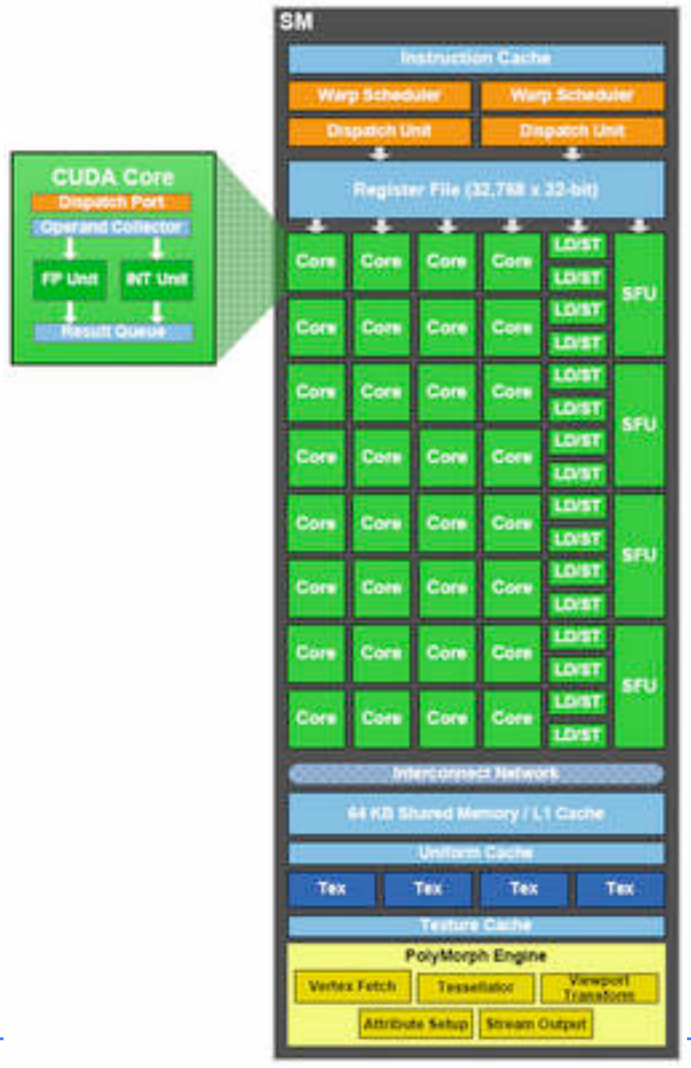
Shared memory

Register memory in each core

Instruction handling/thread management

2010: Fermi (GT100)





16 SMs

32 cores per SM

Important change:

Much area for L2 cache!

More on Fermi

4x performance for double (64-bit FP)

More silicon space for cache! More like a CPU.

CGPU = Computing Graphics Processing Unit

=> NVidia aims for GPGPU with Fermi!

2018: Turing

Big change towards specialized parts

- **Tensor cores**
- **RT cores**
- **Focus on raytracing and learning**

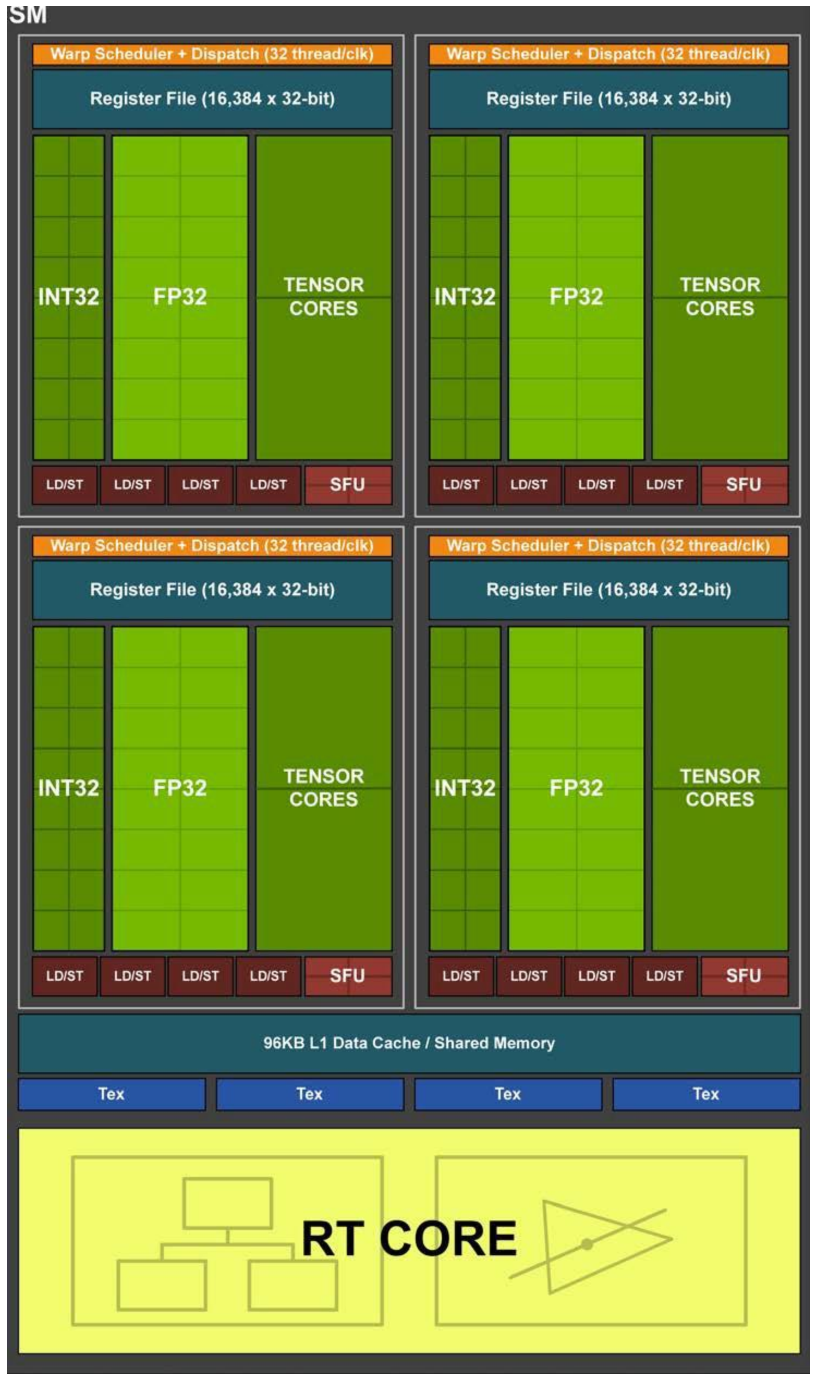
Still new - Is it a big step?

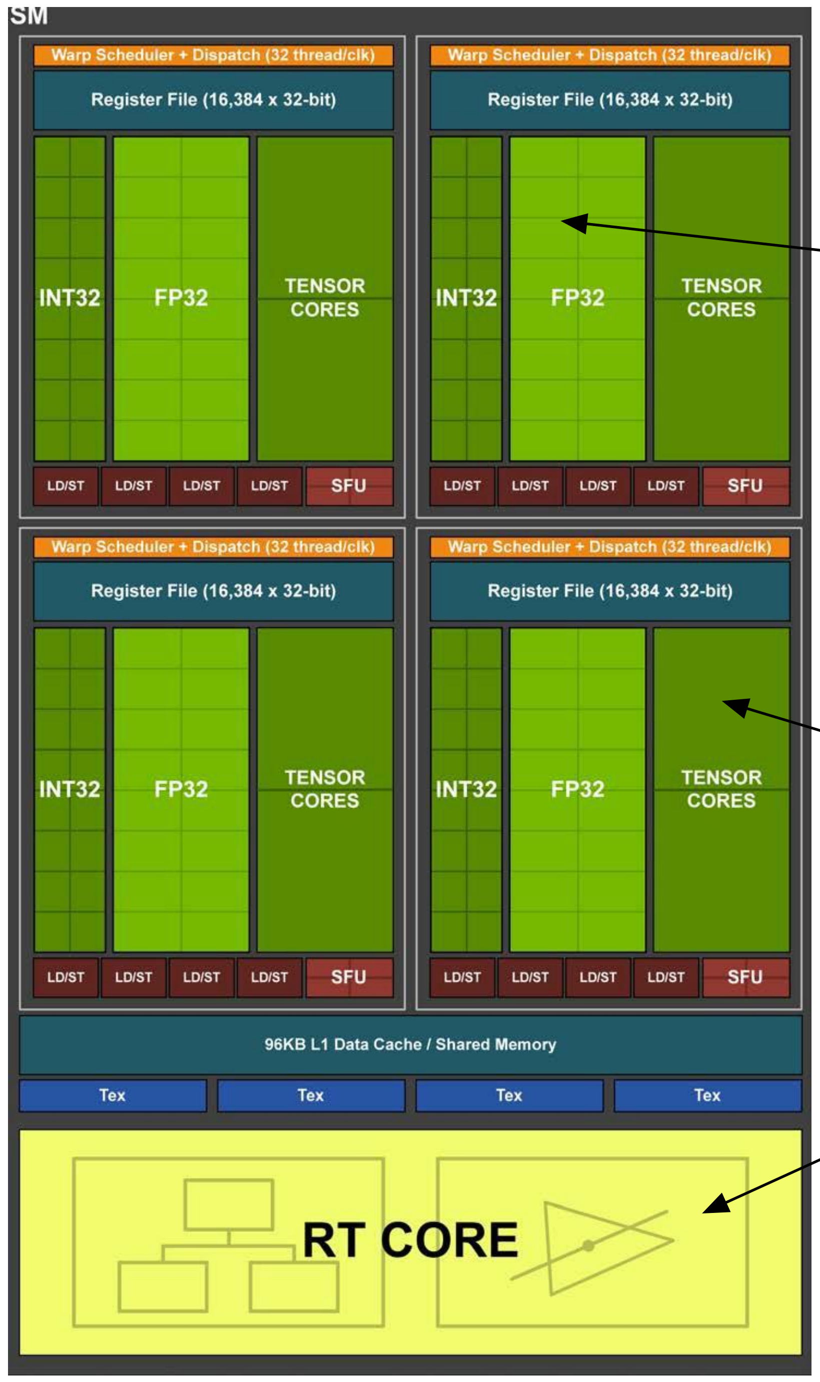
Turing vs G80

G80 = unification, only one kind of cores = better use of hardware

Turing = separation, three kinds of cores... meaning what?

Contradiction! Will this last?


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

General purpose hardware

Special purpose hardware

Questionable usability for general purpose computations

GPU hardware now in three parts

- General purpose
- Real time ray-tracing
- Deep learning

General pupose first, then a look into the others

Turing GPUs

The latest and hottest - and the biggest change since G80!

- RT cores
- Tensor cores
- Cooperative groups
- Modified thread model

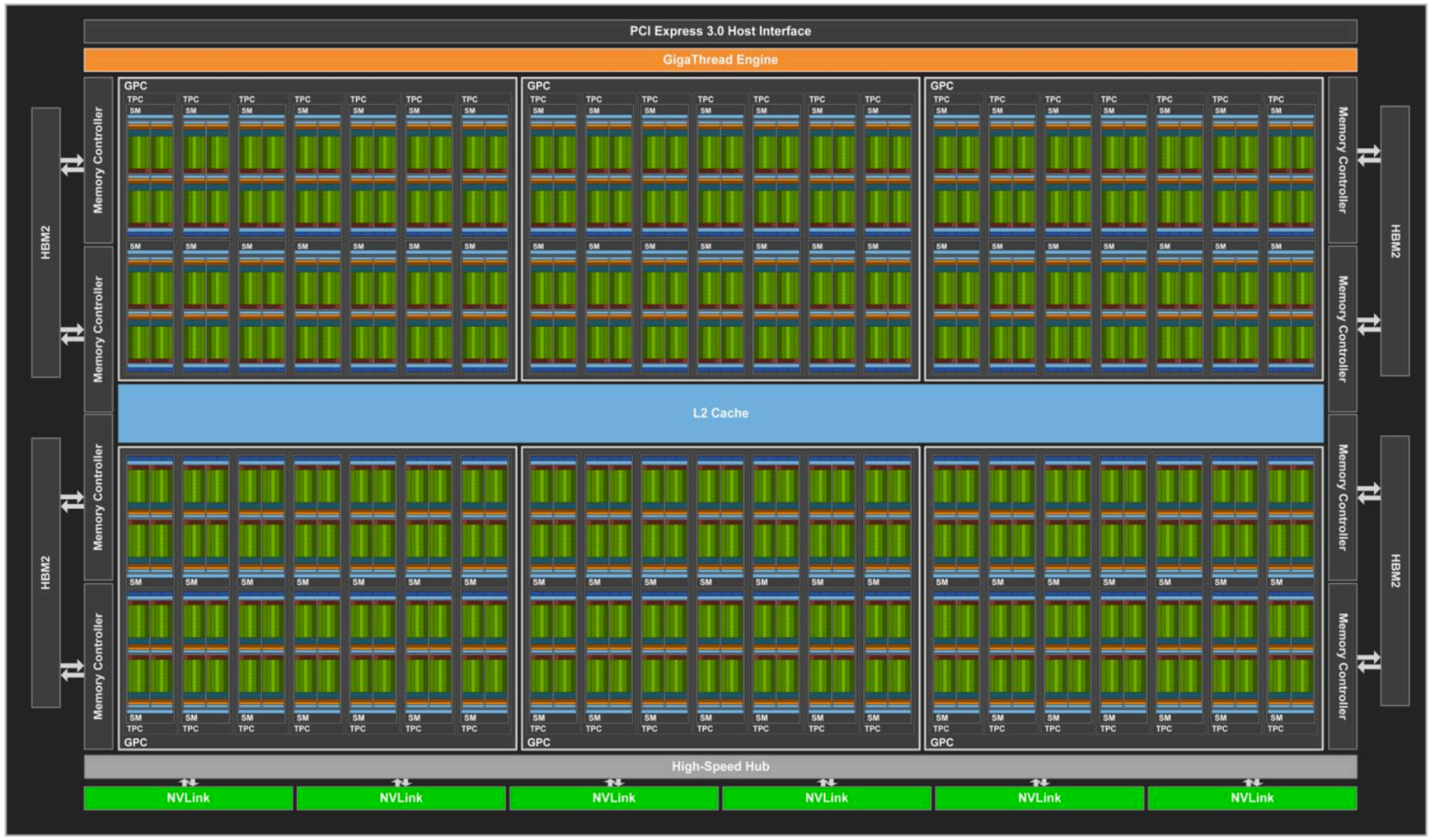
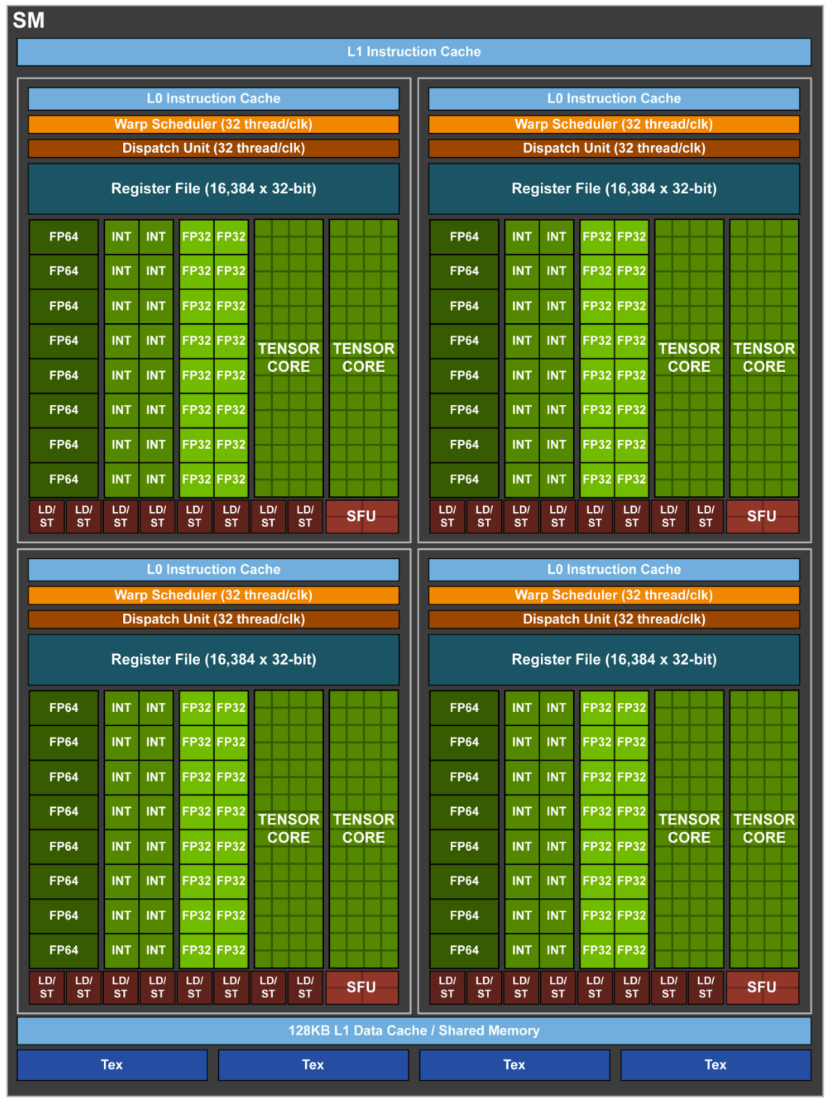


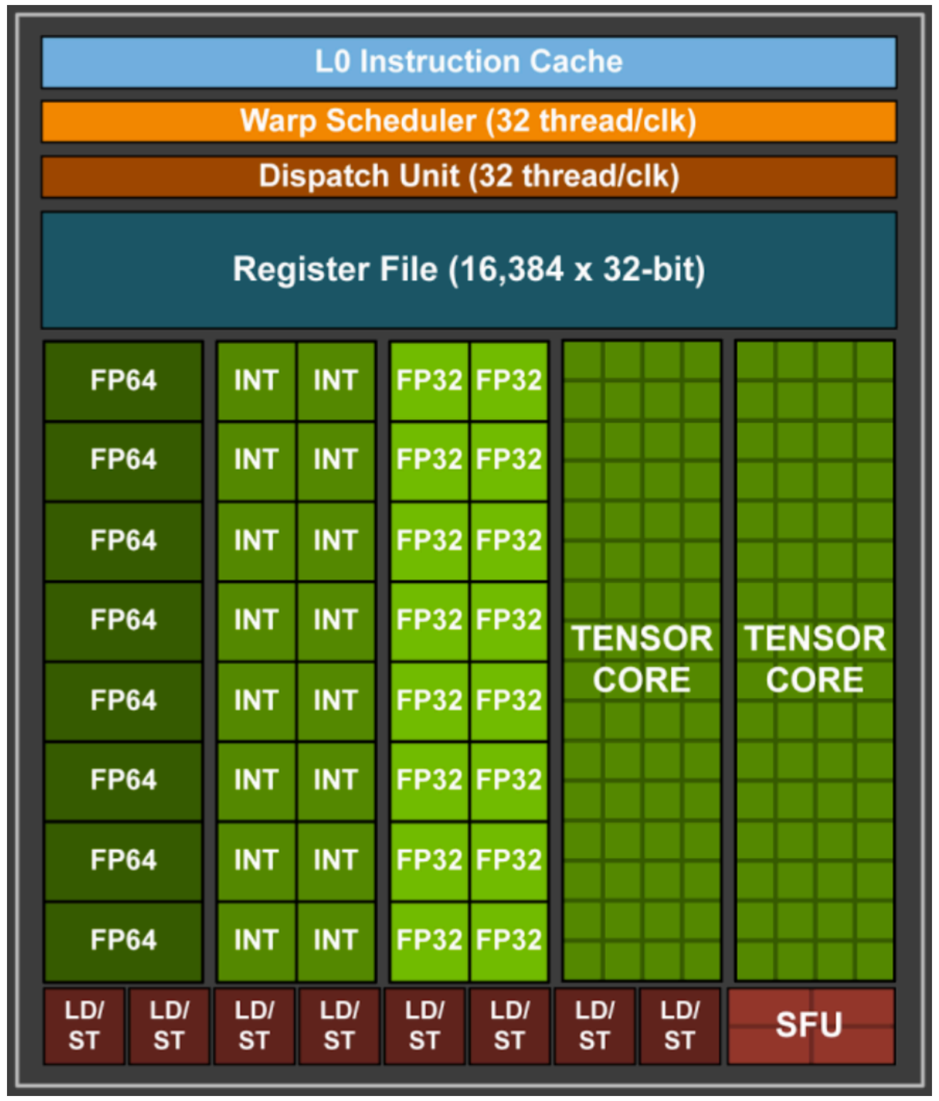
Figure 4. Volta GV100 Full GPU with 84 SM Units

6 groups with
14 SMs in each
= 84 SMs

Multiple levels

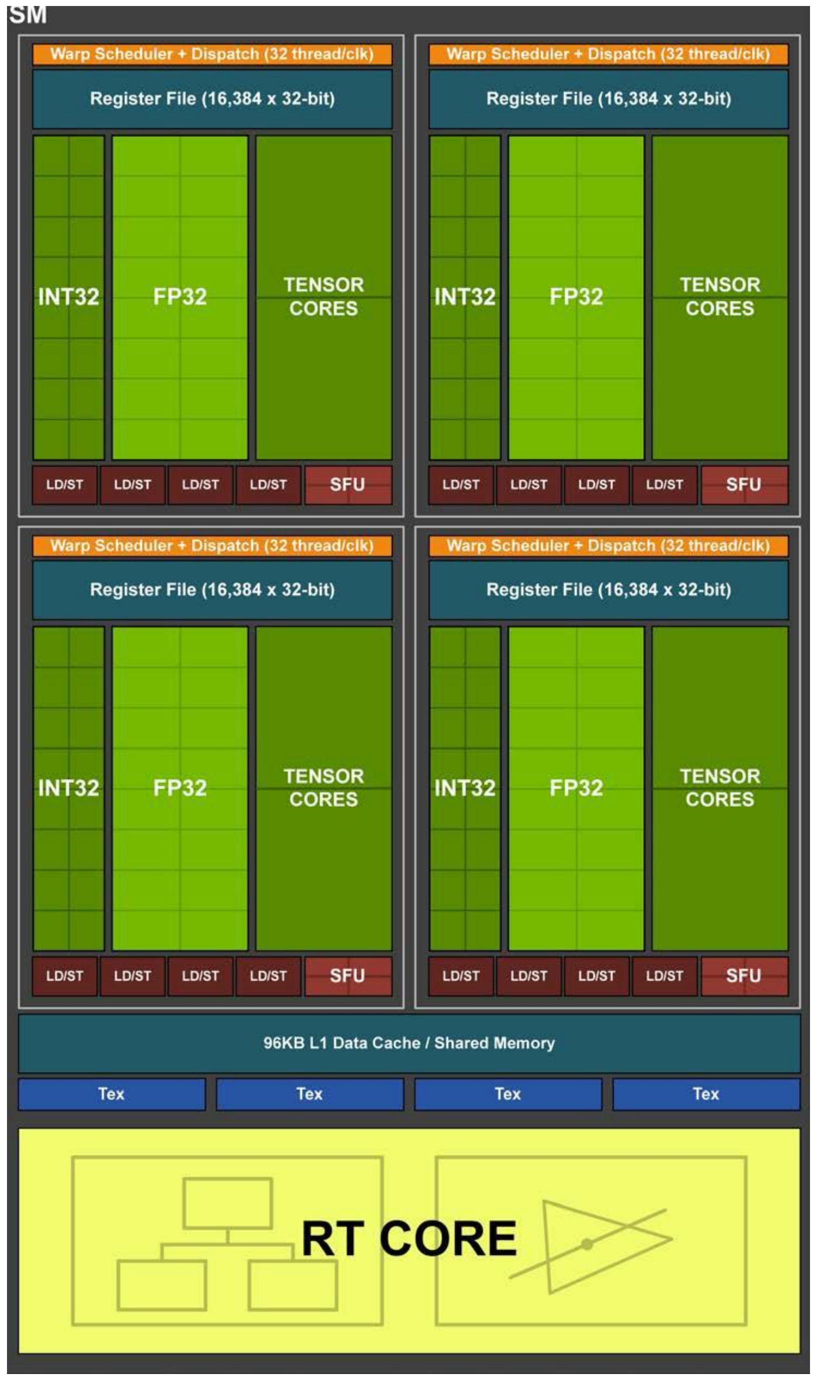
Each SM = 4 different kind of computing cores


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex


L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU

64 FP32 cores
64 INT32 cores
32 FP64 cores
8 Tensor cores

in 4 groups


SM
Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
INT32
FP32
TENSOR CORES
LD/ST
SFU
96KB L1 Data Cache / Shared Memory
Tex
RT CORE

Same as above plus RT core

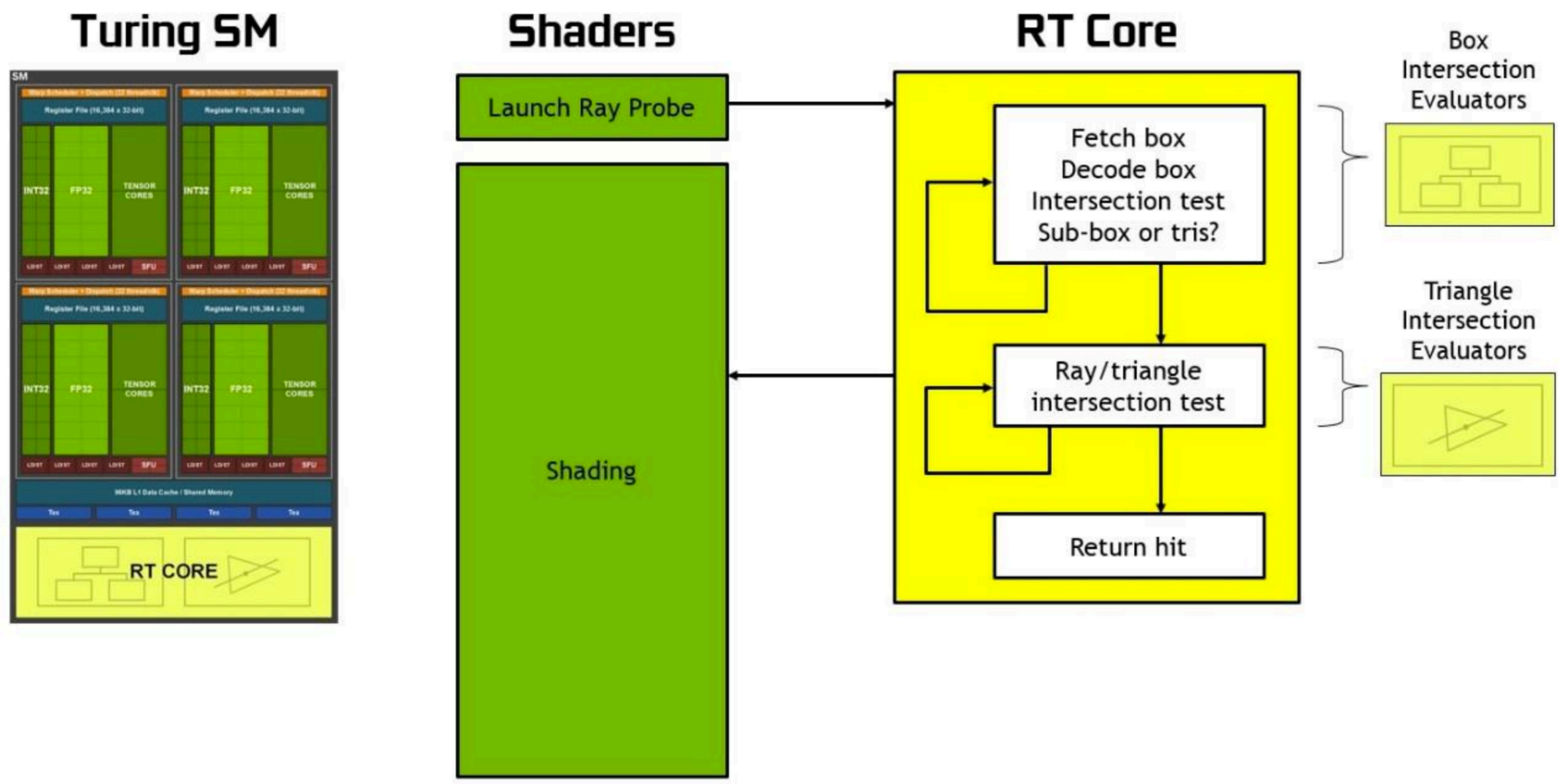
Note that all new hardware is *per SM*!

RT cores = Raytracing cores
Tensor cores

Special-purpose hardware in Turing GPUs

RT cores

Accelerates ray-box and ray-triangle caclulations

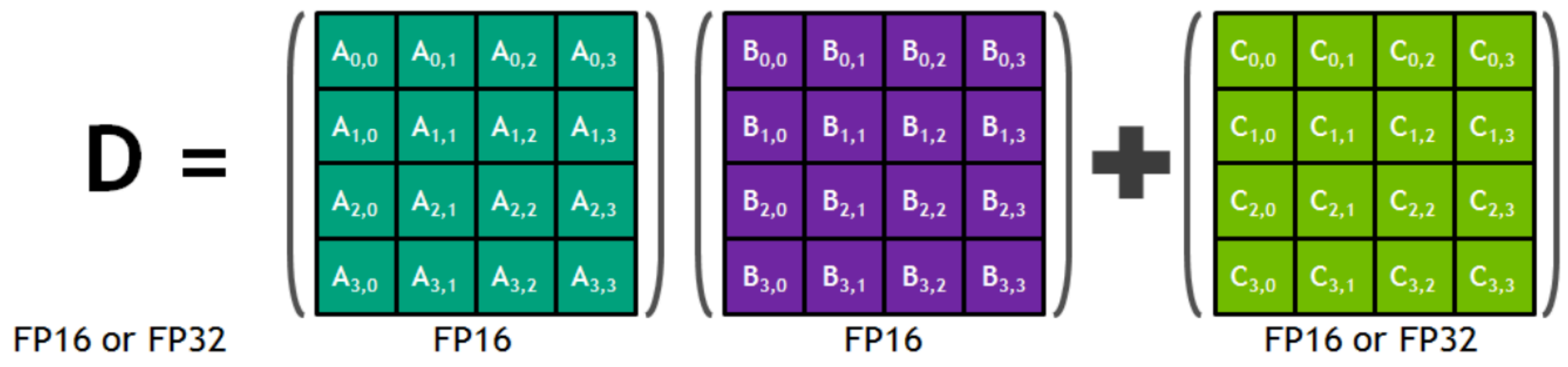


Tensor cores

Accelerates matrix multiply and accumulate

4x4 matrix multiplication

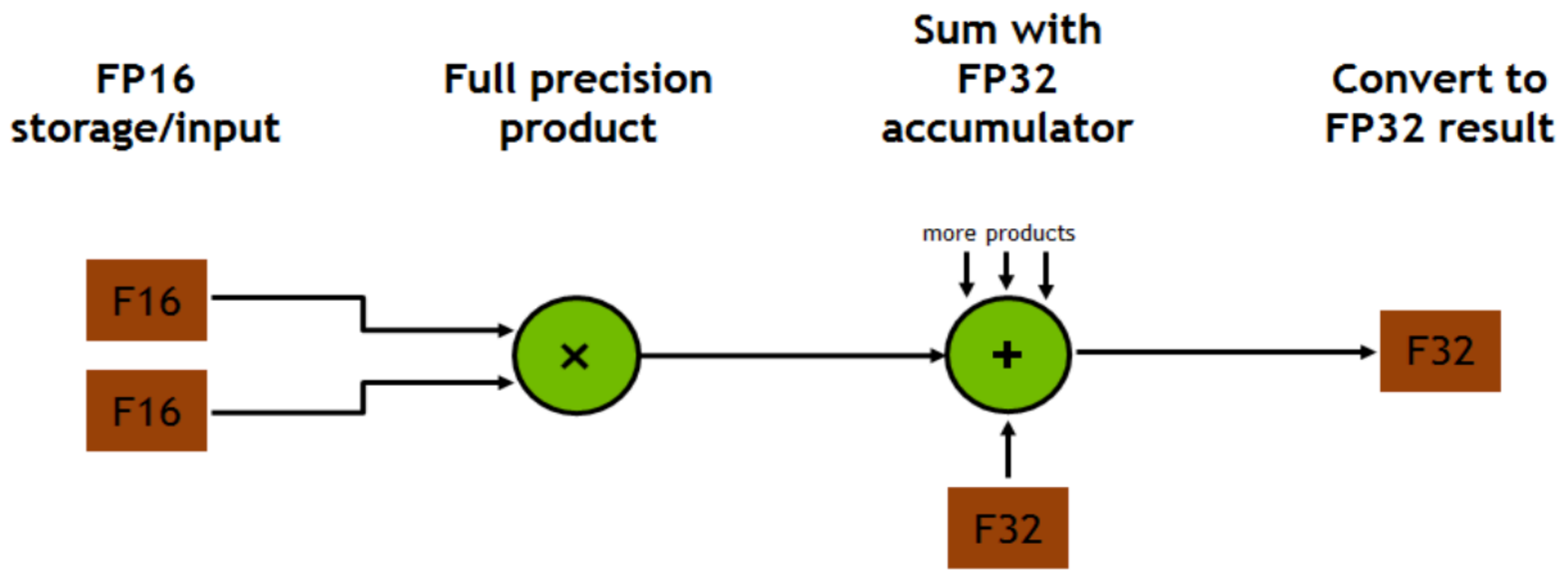
Matrix multiplication in *low precision*



Build bigger multiplications with 4x4 as building block.

Low precision for faster calculation

FP16 in, FP32 out



Using tensor cores

"Inside" CUDA; New subset of CUDA API.

Also used by several libraries.

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void> class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &b, const
fragment<...> &c, bool satf=false);
```

Cooperative groups

Cooperative groups allow synchronization over parts of a block/SM instead of the whole block.

Gives more flexible synchronization, allows more threads to keep working while others wait for a synchronization.

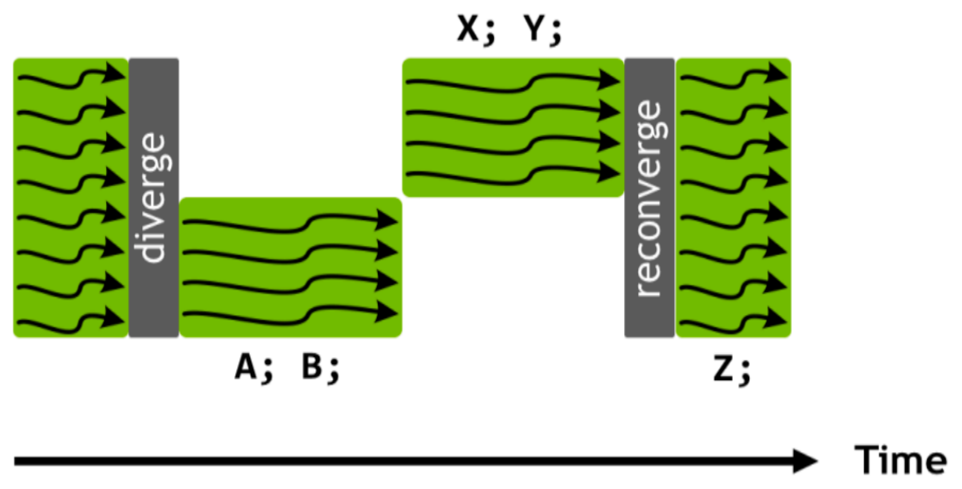
Modified thread model

Thread model: Warps are controlled by an active mask to map out threads depending on branching ("if" statements)

Turing modifies this by interleaving branch execution.

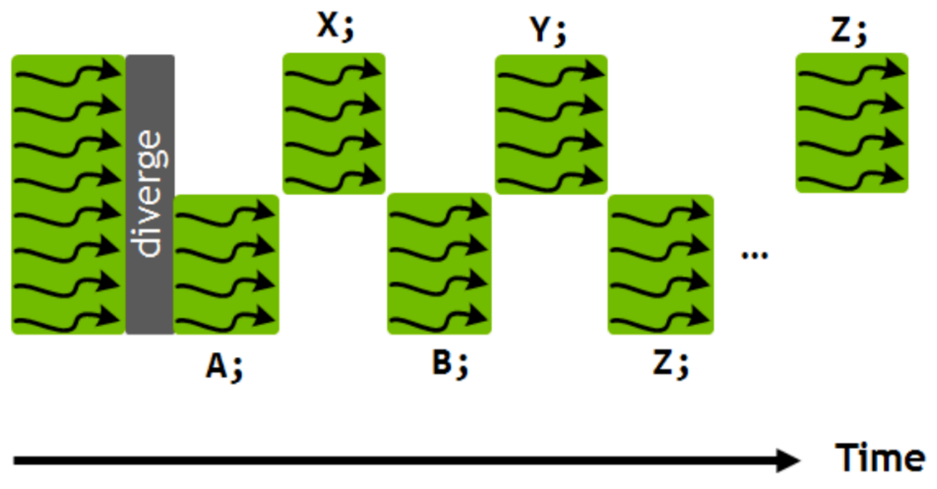
Old model: One branch is executed at a time

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



New model: Branch execution is interleaved

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



Conclusions on Turing:

- Extremely high parallelism: 84 SMs with multiple warp capability and numerous cores in each
- Tensor cores for accelerating matrix mult + accumulate for deep learning
- RT cores
- Additional new flexibility

Related parallelization efforts

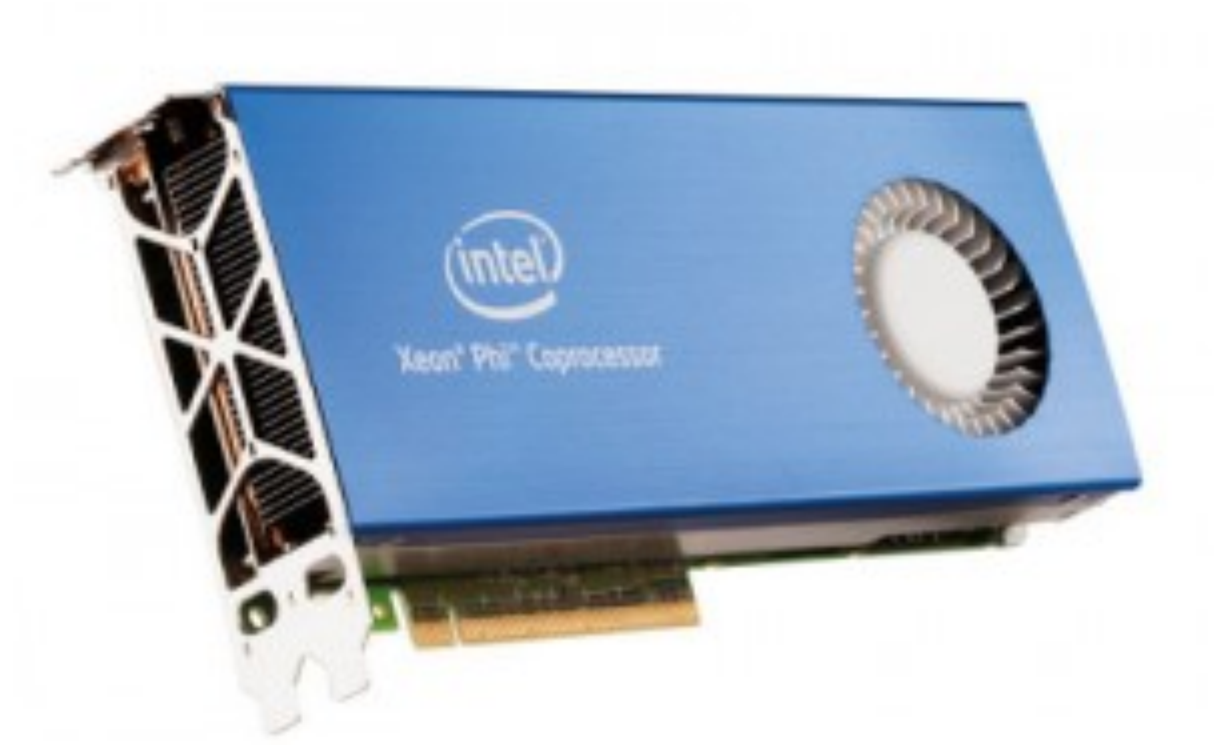
IBM Cell (next generation canceled!)

Intel Larabee (”put on ice” - dead)

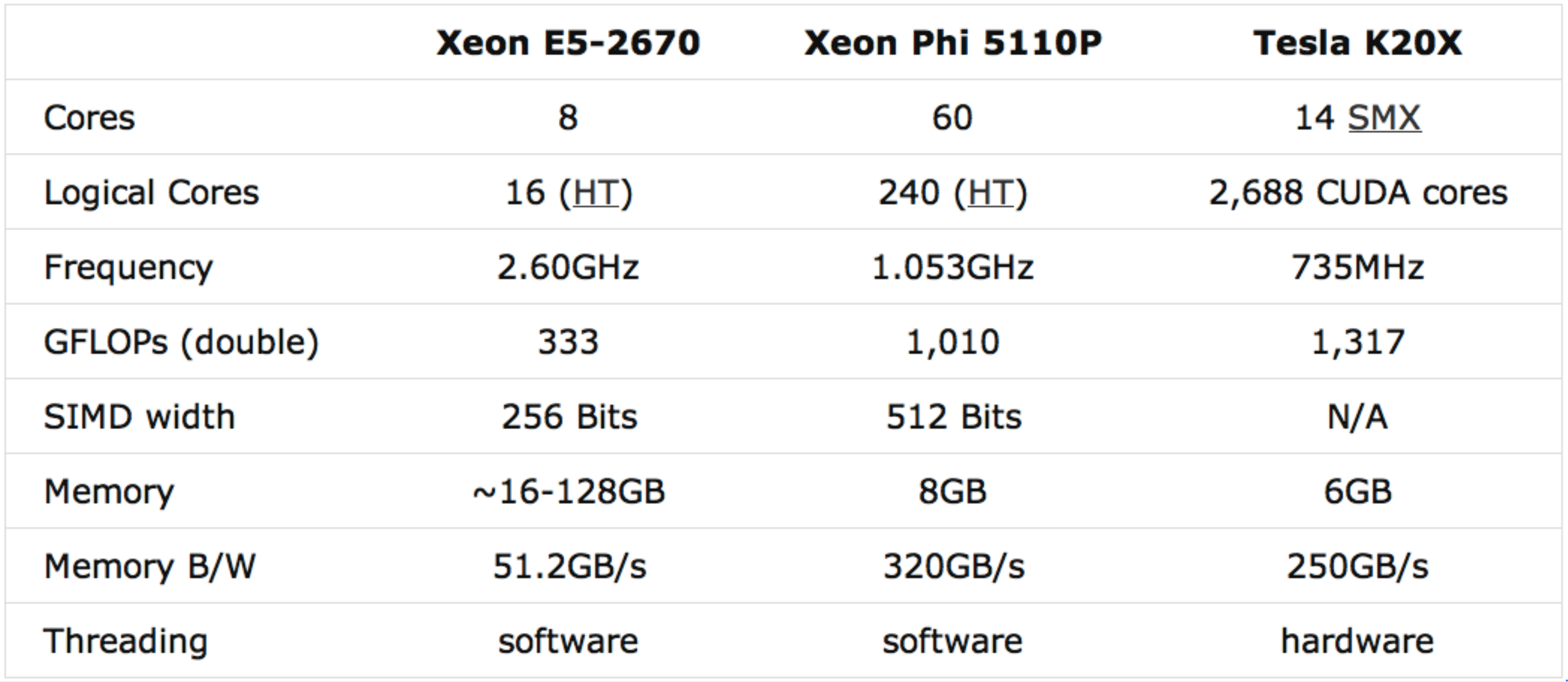
GPUs are the clear winners so far!

But never count out Intel...

how about the more recent Xeon Phi?
(Follow-up on Larabee)

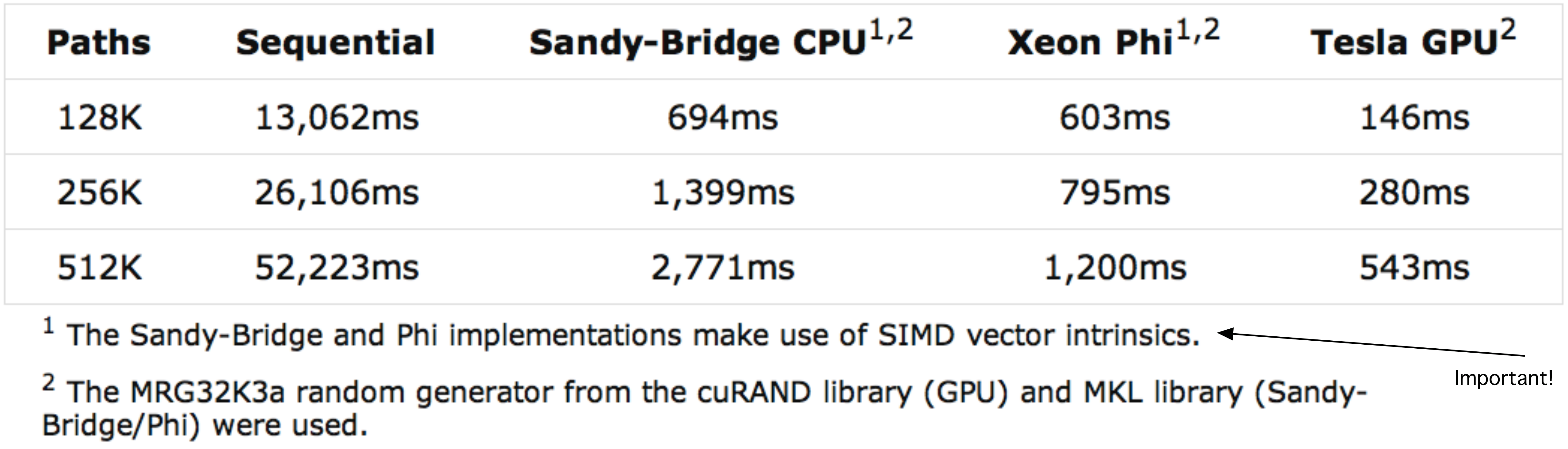


How does it compare?



	Xeon E5-2670	Xeon Phi 5110P	Tesla K20X
Cores	8	60	14 SMX
Logical Cores	16 (HT)	240 (HT)	2,688 CUDA cores
Frequency	2.60GHz	1.053GHz	735MHz
GFLOPs (double)	333	1,010	1,317
SIMD width	256 Bits	512 Bits	N/A
Memory	~16-128GB	8GB	6GB
Memory B/W	51.2GB/s	320GB/s	250GB/s
Threading	software	software	hardware

Test: Does it compete?



Paths	Sequential	Sandy-Bridge CPU[1,2]	Xeon Phi[1,2]	Tesla GPU[2]
128K	13,062ms	694ms	603ms	146ms
256K	26,106ms	1,399ms	795ms	280ms
512K	52,223ms	2,771ms	1,200ms	543ms

[1] The Sandy-Bridge and Phi implementations make use of SIMD vector intrinsics. ← Important!

[2] The MRG32K3a random generator from the cuRAND library (GPU) and MKL library (Sandy-Bridge/Phi) were used.

The GPU still wins! (Even over other SIMD!)

Conclusion comparison
SB - Xeon Phi - GPU

Even the CPU performed pretty well.

All use SIMD (at least partially) for best performance!

All require you to code in parallel!

Dedicated hardware for deep learning

There is work on ASICs for deep learning. Most notable: Tensor processing unit (TPU)? Proprietary, in-house chips. Inflexible.

For flexible, programmable, general-pupose applications, the GPU holds the lead.

And this brought us to:

GPGPU/GPU Computing

General Purpose computation on Graphics Processing Units

Mark Harris, 2002

Perform demanding calculations on the GPU instead of the CPU!

At first, appeared to be a wild idea, but is now a very serious technology! Results were highly varied in the early years, but the GPU advantage has grown bigger and bigger.

GPGPU approaches

- Using fixed pipeline graphics
- Shader programs
- CUDA
- OpenCL
- Compute shaders

Fixed pipeline GPGPU

Reformulate a problem to something that can be done by standard graphics operations.

Limited success 1999/2000. Not of any practical interest!

Example: Jörgen Ahlberg, face tracking

Fragment (pixel) shader based GPGPU

Portable! All GPUs can use shaders, no need for extra software, run using standard software/drivers.

All modern shader languages (GLSL, Cg, HLSL) are similar and easy to program in.

Requires a re-mapping of data to textures.

Very good results already in 2005: 8x speedups overall reported!

CUDA-based GPGPU

Only works on NVidia hardware.

Requires extra software - which isn't very elegant.

Nice integration of CPU and GPU code in the same program.

Excellent results! 100x speedups are common - before optimizing! Even low-end GPUs give significant boosts.

OpenCL-based GPGPU

Works on various hardware - not only GPUs.

Developed by Khronos Group, pushed by Apple.

Harder to get started, software looks pretty much like programming shaders.

OpenGL Compute shaders

Built into OpenGL

Similar to OpenCL

Good portability

Direct Compute Compute shaders

Built into DirectX

Similar to OpenCL

MS only

Vulkan

The "new OpenGL", arrived 2016.

"Bleeding edge".

Future main generic GPU platform for both graphics and computing?

Same compute shaders as OpenGL.

Metal

Apples "Vulkan".

Apple has deprecated everything else - including OpenCL

"Metal Performance Shaders".

Apple only.

Use the source, Luke!

Four trivial examples:

Hello World! for CUDA

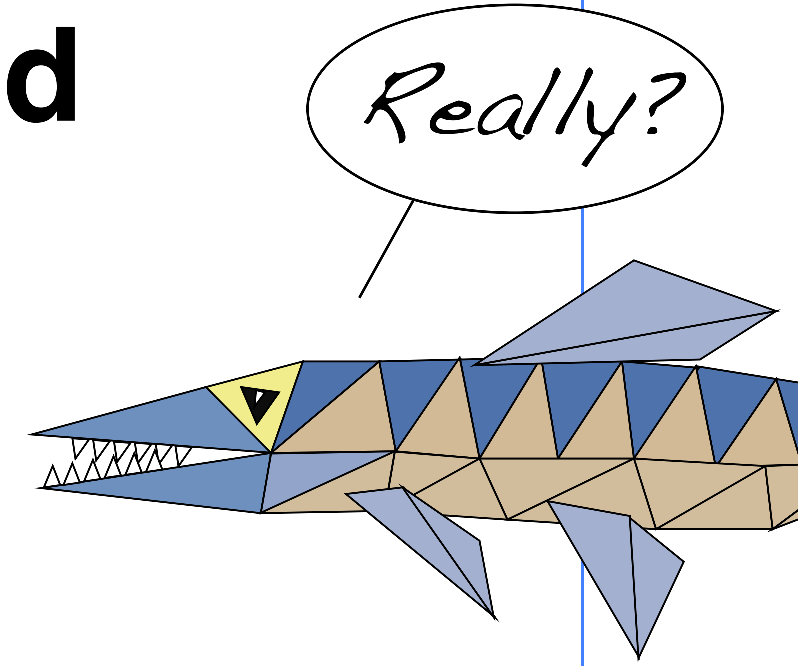
Hello World! for OpenCL

Hello World for GLSL

Hello World for Compute Shaders

Introduction to CUDA

CUDA = Compute Unified Device Architecture



Developed by NVidia

Only available on NVidia boards, G80 or better GPU architecture

Designed to hide the graphics heritage and add control and flexibility

Computing model:

1. Upload data to GPU

2. Execute kernel

3. Download result

Similar to shader-based solutions and OpenCL

Integrated source

Source of host and kernel code in the same source file!

Major difference to shaders and OpenCL.

Kernel code identified by special modifiers.

About CUDA

Architecture and C extension

Spawn a large number of threads, to be ran virtually in parallel

Just like in graphics! Fragments/computations not *quite* executed in parallel.

A bunch at a time - a *warp*.

Looks much more like an ordinary C program! No more ”data stored as pixels” - just arrays!

Simple CUDA example

A working, compilable example

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void simple(float *c)
{
 c[threadIdx.x] = threadIdx.x;
}

int main()
{
 int i;
 float *c = new float[N];
 float *cd;
 const int size = N*sizeof(float);

 cudaMalloc( (void**)&cd, size );
 dim3 dimBlock( blocksize, 1 );
 dim3 dimGrid( 1, 1 );
 simple<<<dimGrid, dimBlock>>>(cd);
 cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
 cudaFree( cd );

 for (i = 0; i < N; i++)
  printf("%f ", c[i]);
 printf("\n");
 delete[] c;
 printf("done\n");
 return EXIT_SUCCESS;
}
```

Simple CUDA example

A working, compilable example

```
#include <stdio.h>

const int N = 16;
const int blocksize = 16;

__global__
void simple(float *c)
{
 c[threadIdx.x] = threadIdx.x;
}

int main()
{
 int i;
 float *c = new float[N];
 float *cd;
 const int size = N*sizeof(float);
```

Kernel

thread identifier

Allocate GPU memory

```
 cudaMalloc( (void**)&cd, size );
 dim3 dimBlock( blocksize, 1 );
 dim3 dimGrid( 1, 1 );
 simple<<<dimGrid, dimBlock>>>(cd);
 cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );
 cudaFree( cd );

 for (i = 0; i < N; i++)
  printf("%f ", c[i]);
 printf("\n");
 delete[] c;
 printf("done\n");
 return EXIT_SUCCESS;
}
```

1 block, 16 threads

Call kernel

Read back data

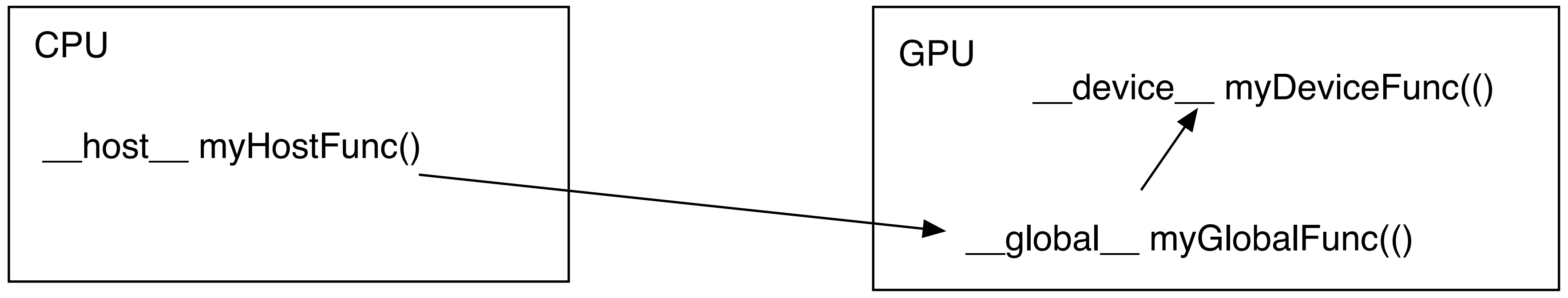
Modifiers for code

Three modifiers are provided to specify how code should be used:

__global__ executes on the GPU, invoked from the CPU. This is the entry point of the kernel.

__device__ is local to the GPU

__host__ is CPU code (superfluous).



Memory management

cudaMalloc(ptr, datasize)
cudaFree(ptr)

Similar to CPU memory management, but done by the CPU to allocate on the GPU

cudaMemCpy(dest, src, datasize, arg)

arg = cudaMemcpyDeviceToHost
or cudaMemcpyHostToDevice

Kernel execution

simple<<<griddim, blockdim>>>(...)

grid = blocks, block = threads

Built-in variables for kernel:

threadIdx* and *blockIdx
blockDim* and *gridDim

(Note that no prefix is used, like GLSL does.)

Compiling Cuda

nvcc

nvcc is nvidia’s tool, /usr/local/cuda/bin/nvcc

Source files suffixed .cu

Command-line for the simple example:

```
nvcc simple.cu -o simple
```

(Command-line options exist for libraries etc)

Compiling Cuda for larger applications

nvcc and gcc in co-operation

nvcc for .cu files

gcc for .c/.cpp etc

Mixing languages possible.

Final linking must include C++ runtime libs.

Example: One C file, one CU file

Example of multi-unit compilation

Source files: cudademokernel.cu and cudademo.c

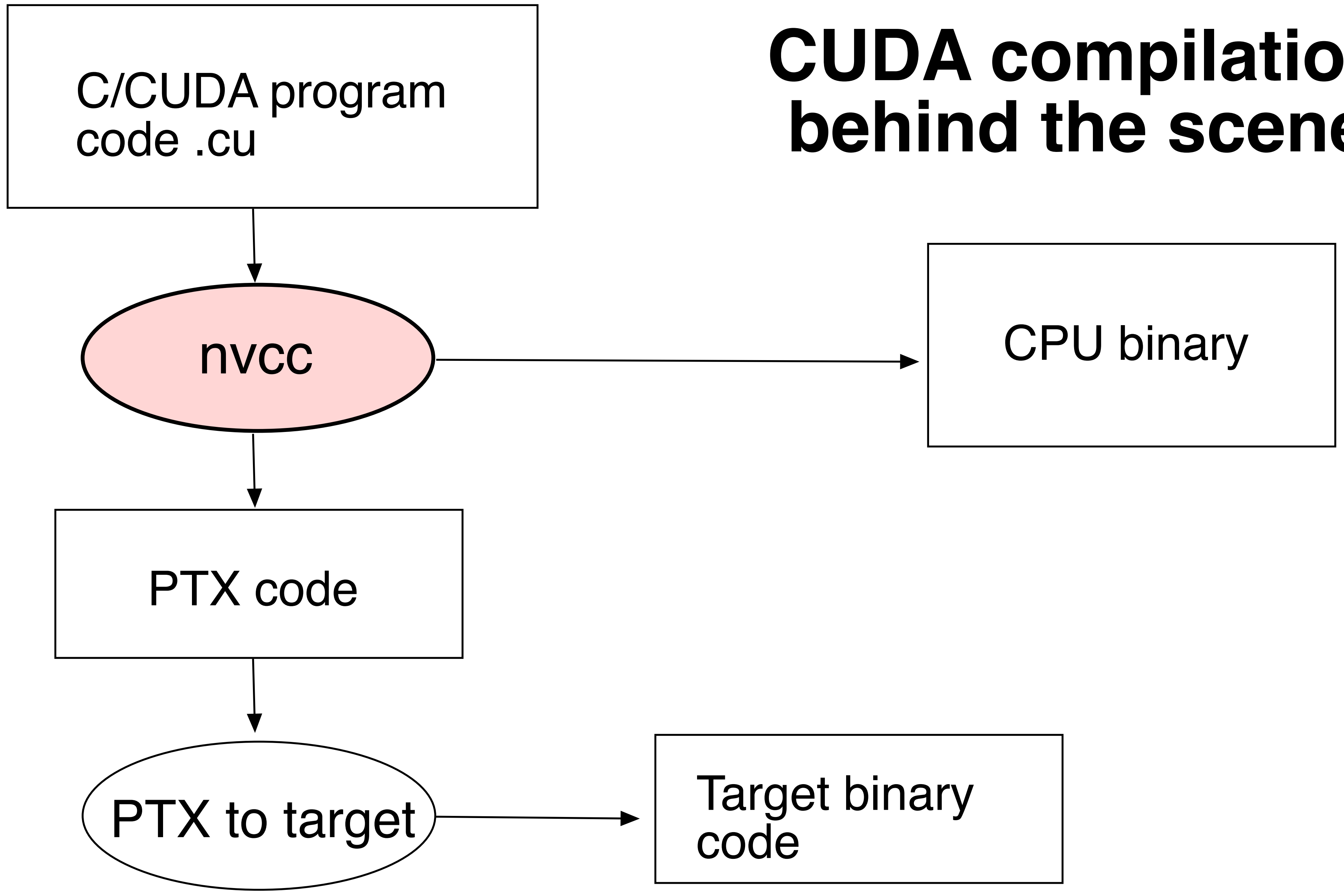
```
nvcc cudademokernel.cu -o cudademokernel.o -c

gcc -c cudademo.c -o cudademo.o -I/usr/local/cuda/include

g++ cudademo.o cudademokernel.o -o cudademo -L/usr/local/
cuda/lib -lcuda -lcudart -lm
```

Link with g++ to include C++ runtime

CUDA compilation behind the scene



Executing a Cuda program

Must set environment variable to find Cuda runtime.

```
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

Then run as usual:

./simple

A problem when executing without a shell!

Launch with execve()

Computing with CUDA

Organization and access

Blocks, threads...

Warps

A warp is the minimum number of data items/threads that will actually be processed in parallel by a CUDA capable device.

We usually don't care about warps but rather discuss threads and blocks.

Processing organization

1 warp = 32 threads

1 kernel - 1 grid

1 grid - many blocks

1 block - 1 SM

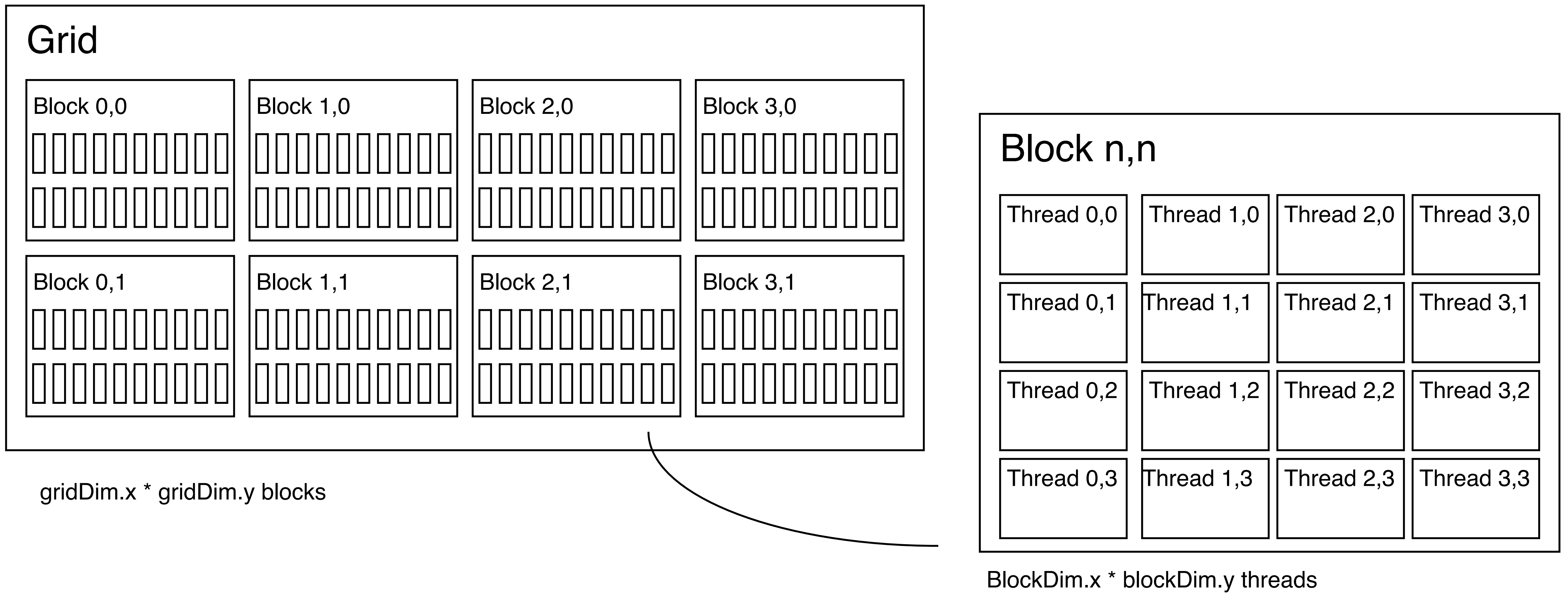
1 block - many threads

Use many threads and many blocks! > 200 blocks recommended.

Thread # multiple of 32

Distributing computing over threads and blocks

Hierarcical model



Indexing data with thread/block IDs

Calculate index by blockIdx, blockDim, threadIdx

Another simple example, calculate square of every element, device part:

```
// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
 int idx = blockIdx.x * blockDim.x + threadIdx.x;
 if (idx<N) a[idx] = a[idx] * a[idx];
}
```

Host part of square example

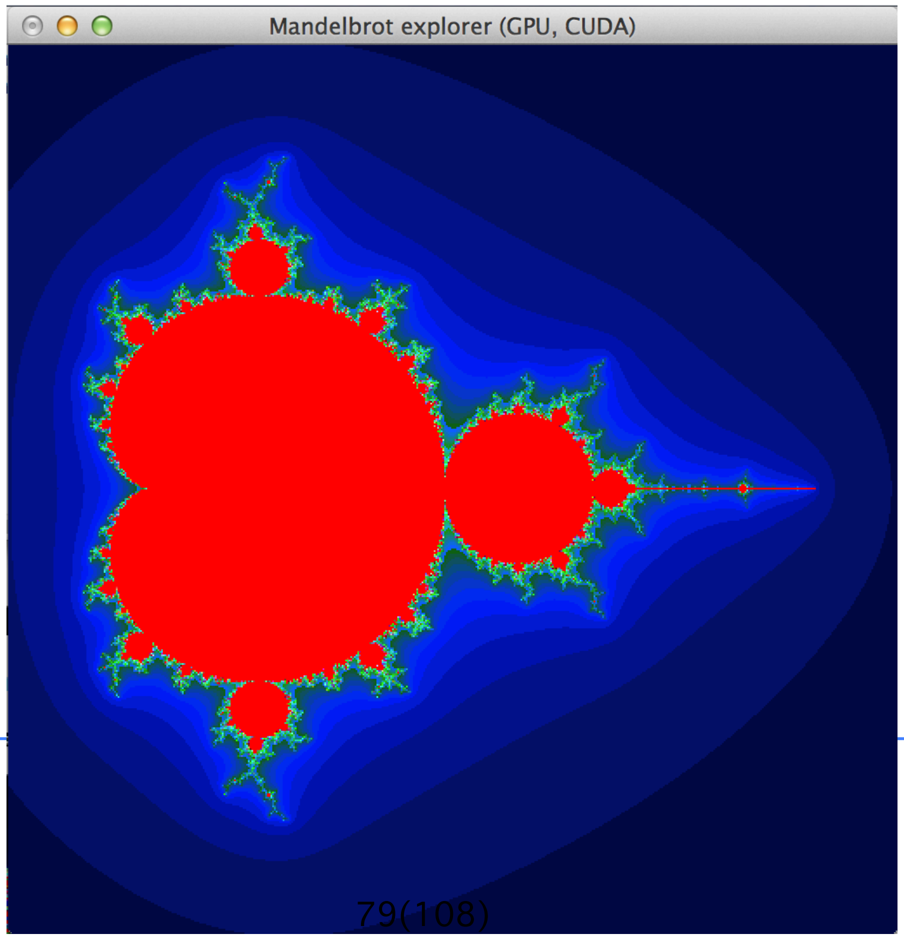
Set block size and grid size

```
// main routine that executes on the host
int main(int argc, char *argv[])
{
 float *a_h, *a_d; // Pointer to host and device arrays
 const int N = 10; // Number of elements in arrays
 size_t size = N * sizeof(float);
 a_h = (float *)malloc(size);
 cudaMalloc((void **) &a_d, size);   // Allocate array on device
// Initialize host array and copy it to CUDA device
 for (int i=0; i<N; i++) a_h[i] = (float)i;
 cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
// Do calculation on device:
 int block_size = 4;
 int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
 square_array <<< n_blocks, block_size >>> (a_d, N);
// Retrieve result from device and store it in host array
 cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
// Print results and cleanup
 for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
 free(a_h); cudaFree(a_d);
}
```

Porting to CUDA: Mandelbrot example

Porting a big computation to the GPU

- **Bigger problem, addressing calculation must be 2D**
- **Simple OpenGL output**



```
void computeFractal( unsigned char *ptr)
{
    // map from x, y to pixel position
    for (int x = 0; x < gImageWidth; x++)
     for (int y = 0; y < gImageHeight; y++)
     {
      int offset = x + y * gImageWidth;

      // now calculate the value at that position
      int fractalValue = mandelbrot( x, y);

      // Colorize it
      int red = 255 * fractalValue/maxiter;
      if (red > 255) red = 255 - red;
      int green = 255 * fractalValue*4/maxiter;
      if (green > 255) green = 255 - green;
      int blue = 255 * fractalValue*20/maxiter;
      if (blue > 255) blue = 255 - blue;

      ptr[offset*4 + 0] = red;
      ptr[offset*4 + 1] = green;
      ptr[offset*4 + 2] = blue;

      ptr[offset*4 + 3] = 255;
     }
}
```

CPU version, for loops

CPU version, multi-threaded?

- **Maybe 8 or 16 threads**

- **Load balancing critical, the work must be distributed among the threads. Non-trivial problem!**

Speedup about 4 times.

```
__global__ void computeFractal( unsigned char *ptr, float scale, float offsetx, float offsety, int imageWidth, int imageHeight, int maxiter)
{
   // map from blockIdx to pixel position
   int x = blockIdx.x * blockDim.x + threadIdx.x;
   int y = blockIdx.y * blockDim.y + threadIdx.y;
   int offset = x + y * gridDim.x * blockDim.x;

     // now calculate the value at that position
     MYFLOAT jx = scale * (MYFLOAT)(imageWidth/2 - x + offsetx/scale)/(imageWidth/2);
     MYFLOAT jy = scale * (MYFLOAT)(imageHeight/2 - y + offsety/scale)/(imageWidth/2);
     int fractalValue = mandelbrot( jx, jy, maxiter);

     // Colorize it
     int red = 255 * fractalValue/maxiter;
     if (red > 255) red = 255 - red;
     int green = 255 * fractalValue*4/maxiter;
     if (green > 255) green = 255 - green;
     int blue = 255 * fractalValue*20/maxiter;
     if (blue > 255) blue = 255 - blue;

     ptr[offset*4 + 0] = red;
     ptr[offset*4 + 1] = green;
     ptr[offset*4 + 2] = blue;

     ptr[offset*4 + 3] = 255;
}
```

GPU version

- **Replace for loops by threads**
- **One thread per pixel!**

Mandelbrot conclusions

Many blocks, many treads in each block. Make sure everything is in use.

Index by thread and block.

Exceptional speedup - trivially parallellizable problem!

Load balancing? No problem. Why?

Conclusion about indexing

Every thread does its own calculation for indexing memory!

blockIdx, blockDim, threadIdx

1, 2 or 3 dimensions

Usually 2 dimensions

Let us talk about optimizations...

For most problems, the threads are not independent and need to access much data!

Memory access

Vital for performance!

Memory types

Coalescing

Example of using shared memory

Memory types

Global

Shared

Constant (read only)

Texture cache (read only)

Local

Registers

Care about these when optimizing - not to begin with

Global memory

400-600 cycles latency!

Shared memory fast temporary storage

Coalesce memory access!

Continuous
Aligned on power of 2 boundary
Addressing follows thread numbering

Use shared memory for reorganizing data for coalescing!

Using shared memory to reduce number of global memory accesses

Read blocks of data to shared memory

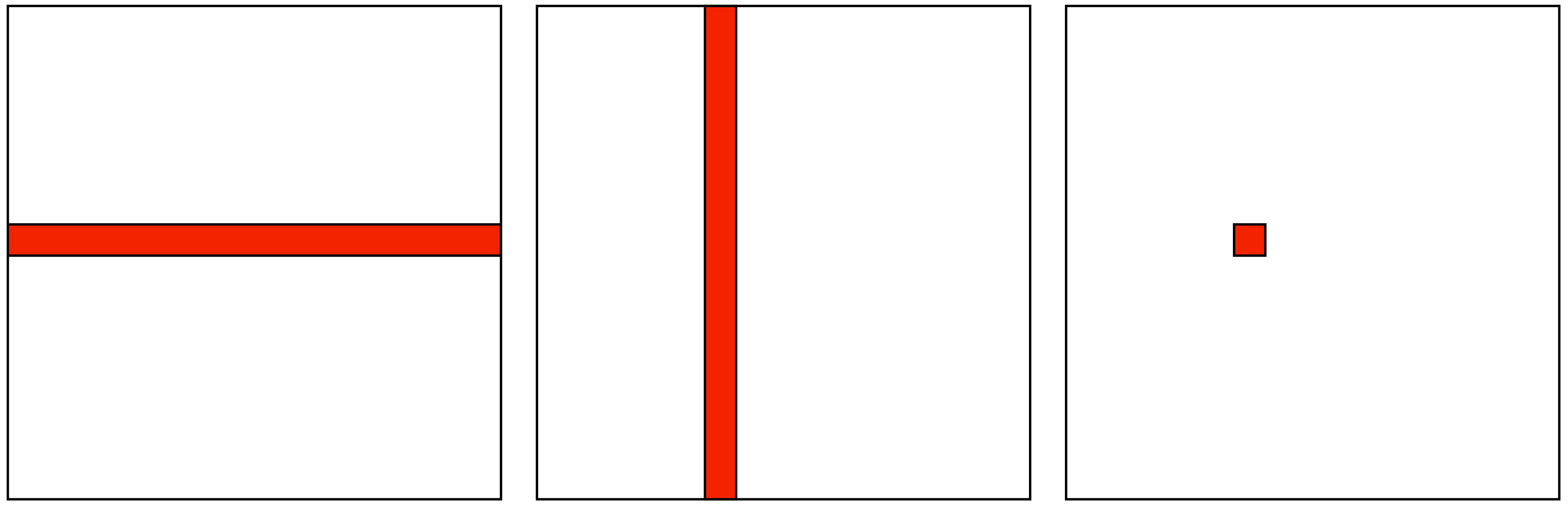
Process

Write back as needed

Shared memory as ”manual cache”

Example: Matrix multiplication

Matrix multiplication



To multiply two N*N matrices, every item will have to be accessed N times!

Naive implementation: $2N^3$ global memory accesses!

Matrix multiplication on CPU

Simple triple ”for” loop

```
void MatrixMultCPU(float *a, float *b, float *c, int theSize)
{
 int sum, i, j, k;

 // For every destination element
 for(i = 0; i < theSize; i++)
  for(j = 0; j < theSize; j++)
  {
   sum = 0;
   // Sum along a row in a and a column in b
   for(k = 0; k < theSize; k++)
    sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
   c[i*theSize + j] = sum;
  }
}
```

Naive GPU version

Replace outer loops by thread indices

```
__global__ void MatrixMultNaive(float *a, float *b, float *c, int
theSize)
{
 int sum, i, j, k;

 i = blockIdx.x * blockDim.x + threadIdx.x;
 j = blockIdx.y * blockDim.y + threadIdx.y;

 // For every destination element
 sum = 0;
 // Sum along a row in a and a column in b
 for(k = 0; k < theSize; k++)
  sum = sum + (a[i*theSize + k]*b[k*theSize + j]);
 c[i*theSize + j] = sum;
}
```

Naive GPU version inefficient

Every thread makes 2N global memory accesses!

Can be significantly reduced using shared memory

Optimized GPU version

Data split into one output patch per block.

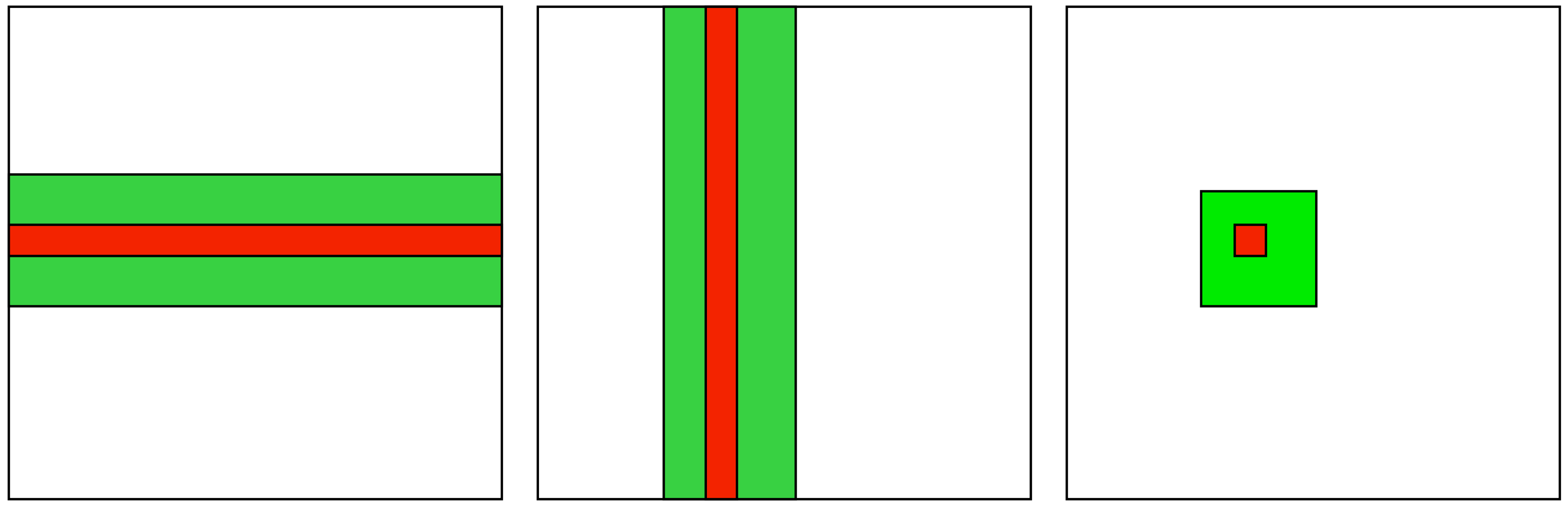
Every element takes part in all the blocks in the same *row* for A, *column* for B

For every such block

Every thread reads *one* element to shared memory

Then loop over the appropriate row and column for the block

Contributing areas for patch



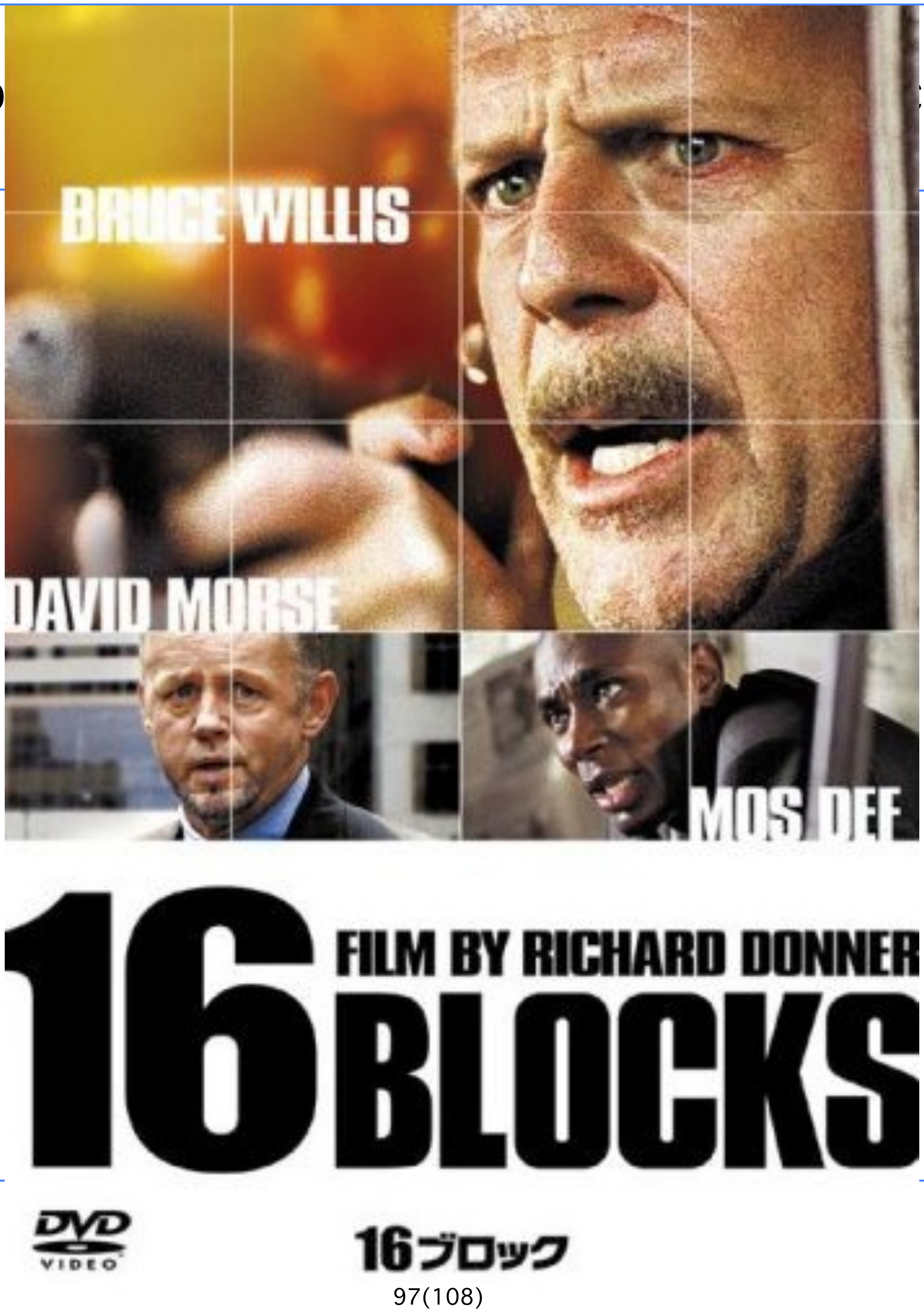
Let each block handle a part of the output (green right).

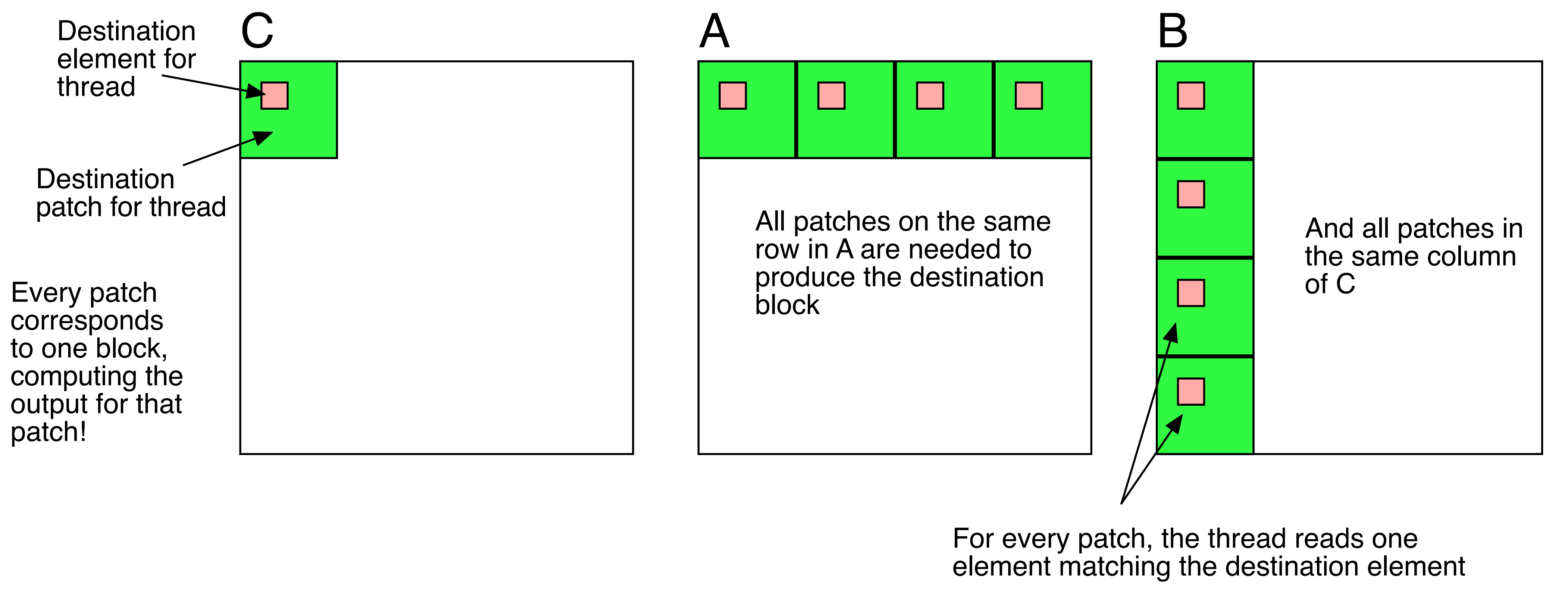
Green areas middle and left contibute to output.

Load the contributing areas into shared memory.

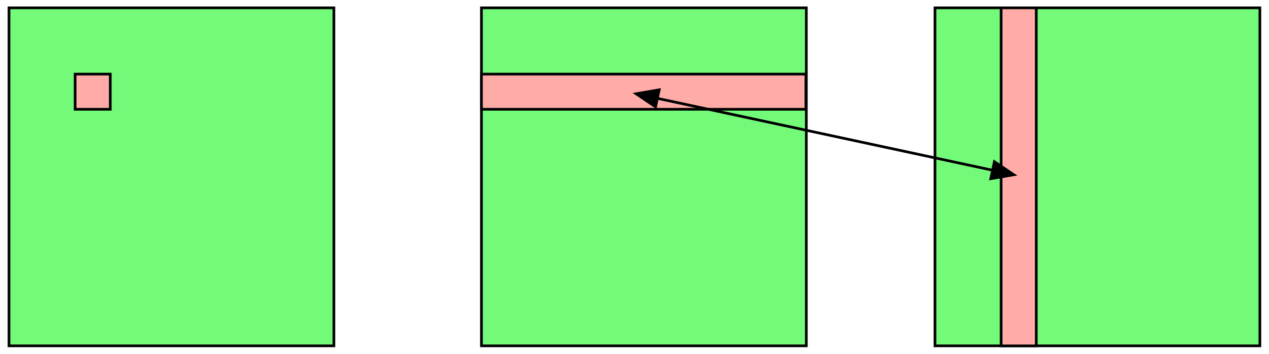
Example: 16 blocks

Info s, ISY, LiTH



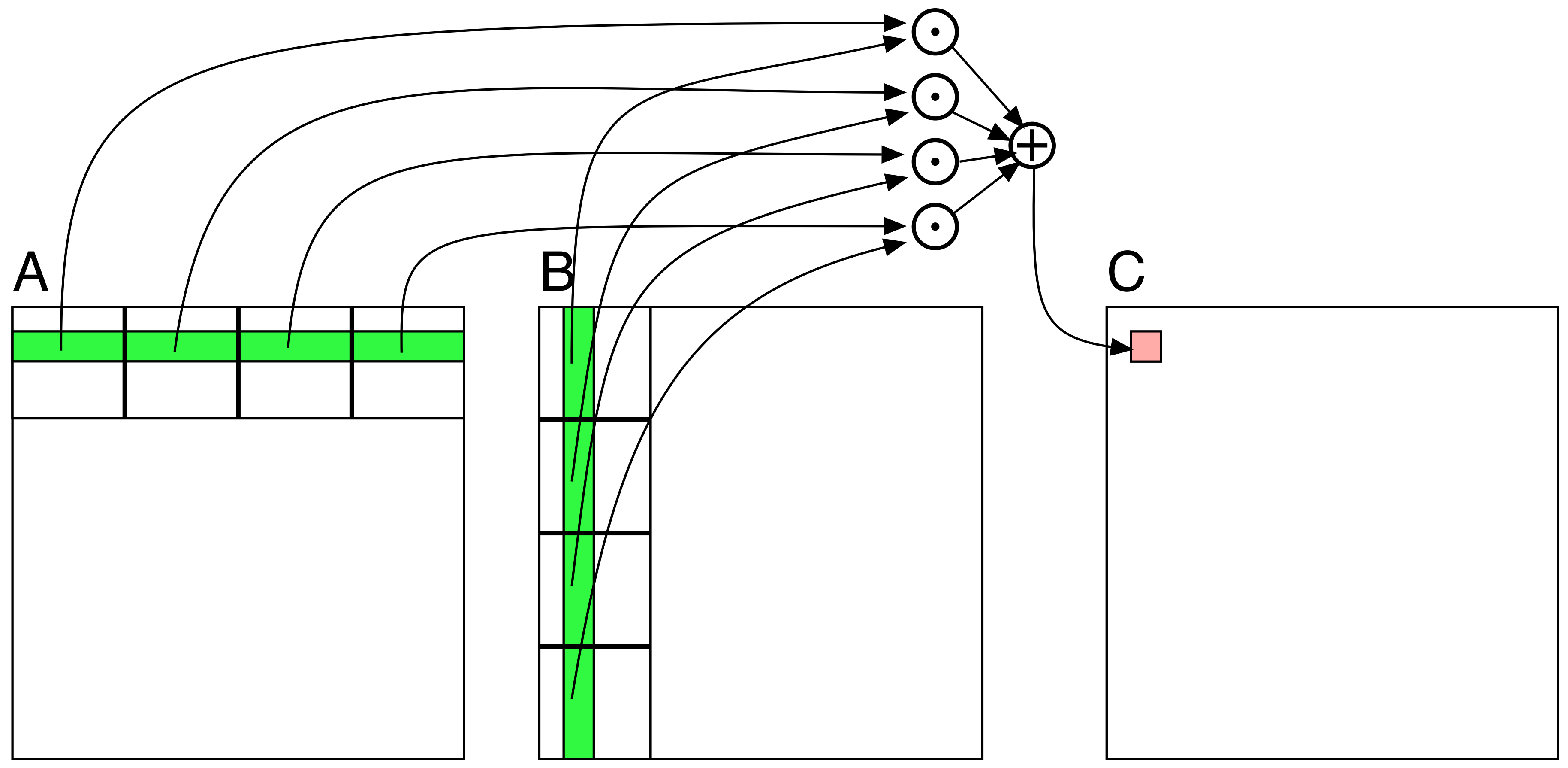

Destination element for thread
C
Destination patch for thread
Every patch corresponds to one block, computing the output for that patch!
A
All patches on the same row in A are needed to produce the destination block
B
And all patches in the same column of C
For every patch, the thread reads one element matching the destination element

For every patch, we loop over the part of one row and column to perform that part of the computation



What one thread reads is used by everybody in the same row (A) or column (B)!

Piece by piece, patch by patch



Optimized GPU version

Loop over patches (1D)

Allocate shared memory

Copy one element to shared memory

Loop over row/column in patch, compute, accumulate result for one element

Write result to global memory

```
__global__ void MatrixMultOptimized( float* A, float* B, float* C, int theSize)
{
 int i, j, k, b, ii, jj;

// Global index for thread
 i = blockIdx.x * blockDim.x + threadIdx.x;
 j = blockIdx.y * blockDim.y + threadIdx.y;

 float sum = 0.0;
 // for all source patches
 for (b = 0; b < gridDim.x; b++)
 {
  __shared__ float As[BLOCKSIZE*BLOCKSIZE];
  __shared__ float Bs[BLOCKSIZE*BLOCKSIZE];

  // Index locked to patch
  ii = b * blockDim.x + threadIdx.x;
  jj = b * blockDim.y + threadIdx.y;

  As[threadIdx.y*blockDim.x + threadIdx.x] = A[ii*theSize + j];
  Bs[threadIdx.y*blockDim.x + threadIdx.x] = B[i*theSize + jj];

  __syncthreads(); // Synchronize to make sure all data is loaded

  // Loop, perform computations in patch
  for (k = 0; k < blockDim.x; ++k)
   sum += As[threadIdx.y*blockDim.x + k]
   * Bs[k*blockDim.x + threadIdx.x];

  __syncthreads(); // Synch so nobody starts next pass prematurely
 }

 C[i*theSize + j] = sum;
}
```

5-10 times faster? So what did I do?

- Decent number of threads and blocks
- Use shared memory for temporary storage
- All threads read ONE item per matrix, but use many!
- Synchronize
- Even more for CPU - compared to single-thread CPU :)

Modified computing model:

Upload data to global GPU memory

For a number of parts, do:

Upload partial data to shared memory

Process partial data

Write partial data to global memory

Download result to host

Synchronization

As soon as you do something where one part of a computation depends on a result from another thread, you must synchronize!

__syncthreads()

Typical implementation:

- **Read to shared memory**
- **__syncthreads()**
- **Process shared memory**
- **__synchthreads()**
- **Write result to global memory**

Synchronization

Really wonderfully simple - everybody are doing the same thing anyway!

Synchronization simply means "wait until everybody are done with this part"

Deadlocks can still occur!

Limitation of synchronization

Synchronization can only be done within a block!
No synchronization between blocks!

Why is this a necessary limitation?

Limitation of synchronization

Synchronization can only be done within a block!
No synchronization between blocks!

Why is this a necessary limitation?

Because all blocks are not active at the same time!
Blocks are queued until an SM is free!

Limitation of synchronization

Synchronization can only be done within a block!
No synchronization between blocks!

Why is this a necessary limitation?

Because all blocks are not active at the same time!
Blocks are queued until an SM is free!

But I *must* synchronize globally!

Answer: Run multiple kernels! More on this later.

Summary:

- Make threads and blocks to make the hardware occupied
- Access data depending on thread/block number
- Memory accesses are expensive!
- Shared memory is fast
- Make threads within a block cooperate
- Synchronize