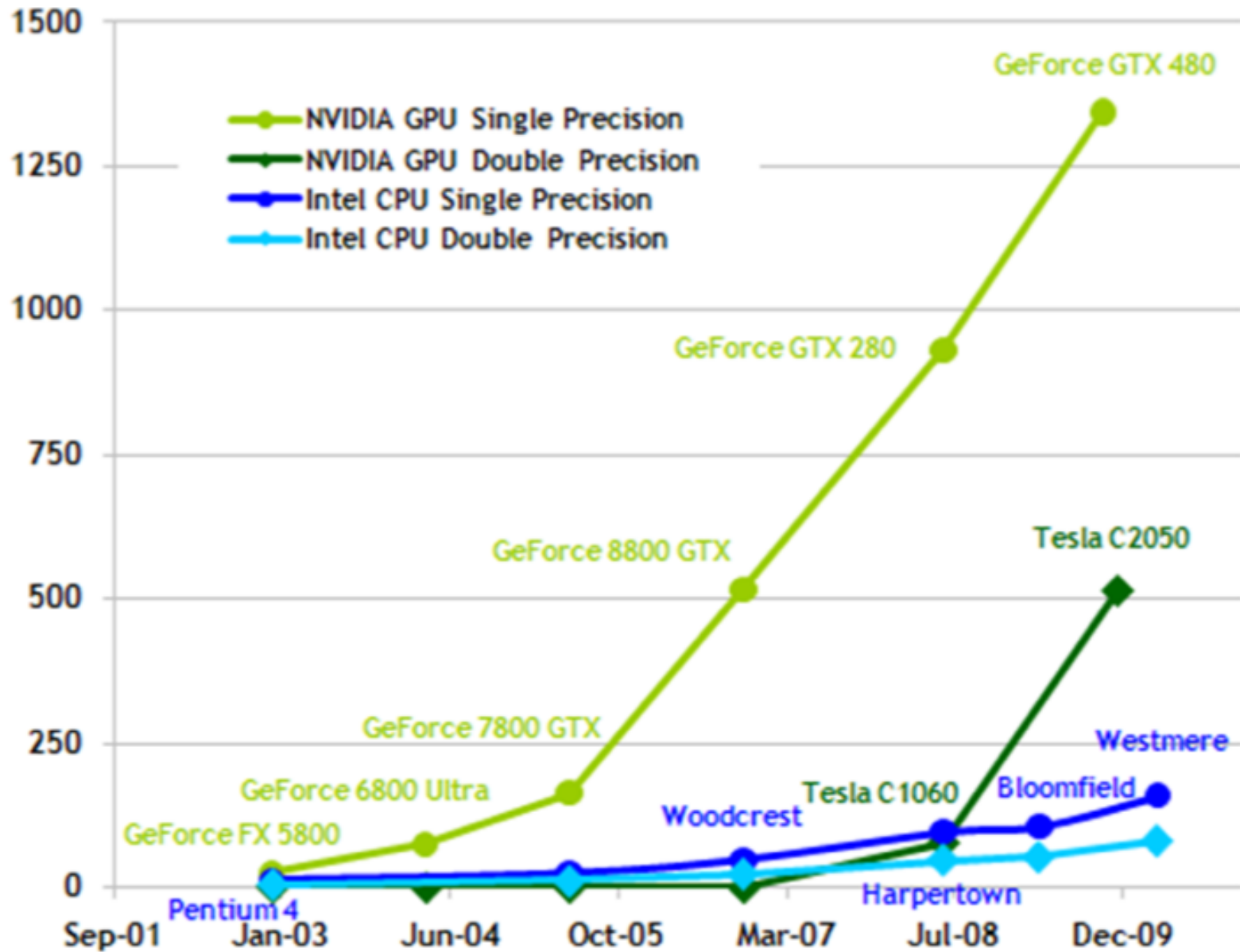


Programming GPUs (CUDA and OpenCL) Part 1: architecture

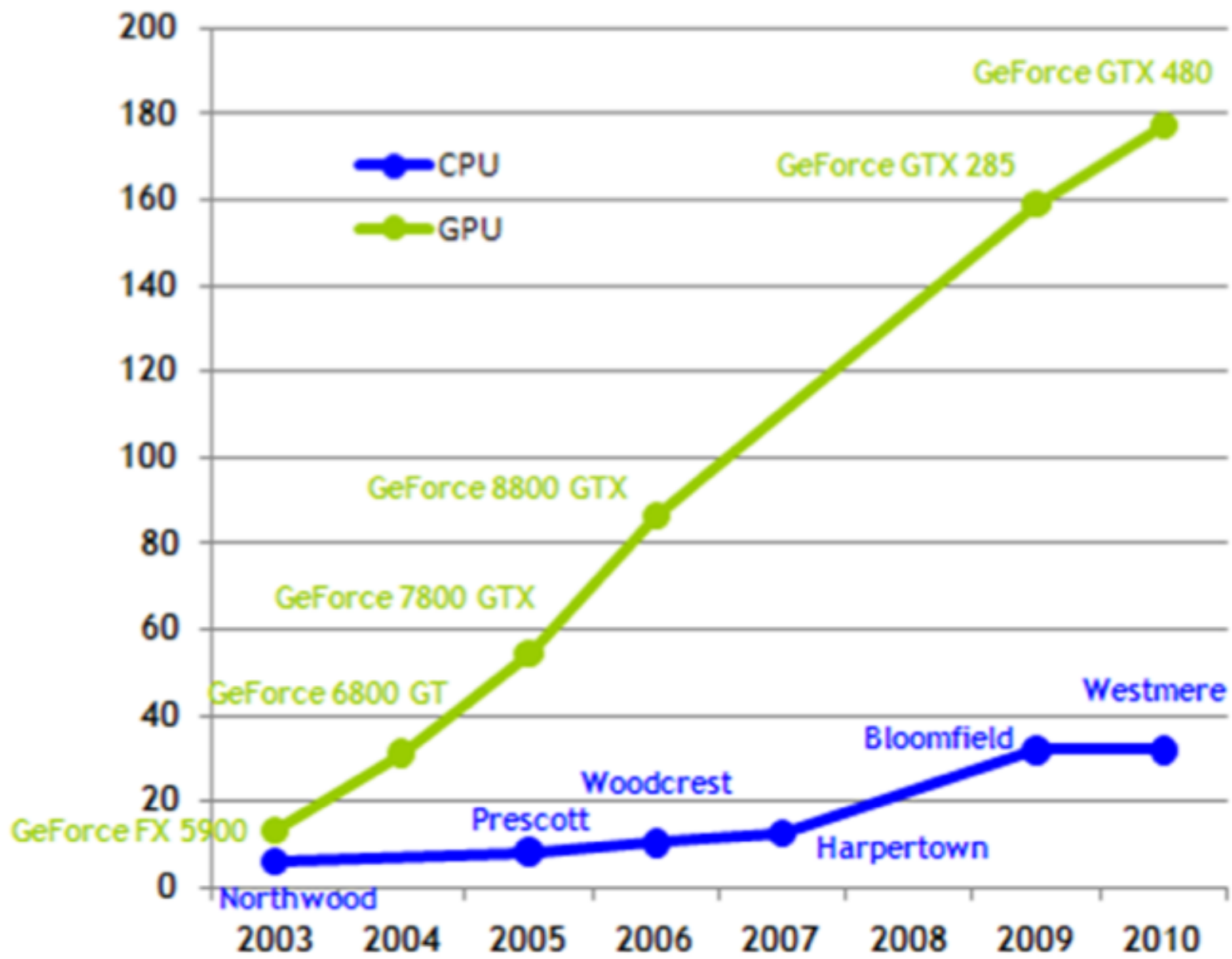
Gordon Erlebacher

Nov. 8, 2010

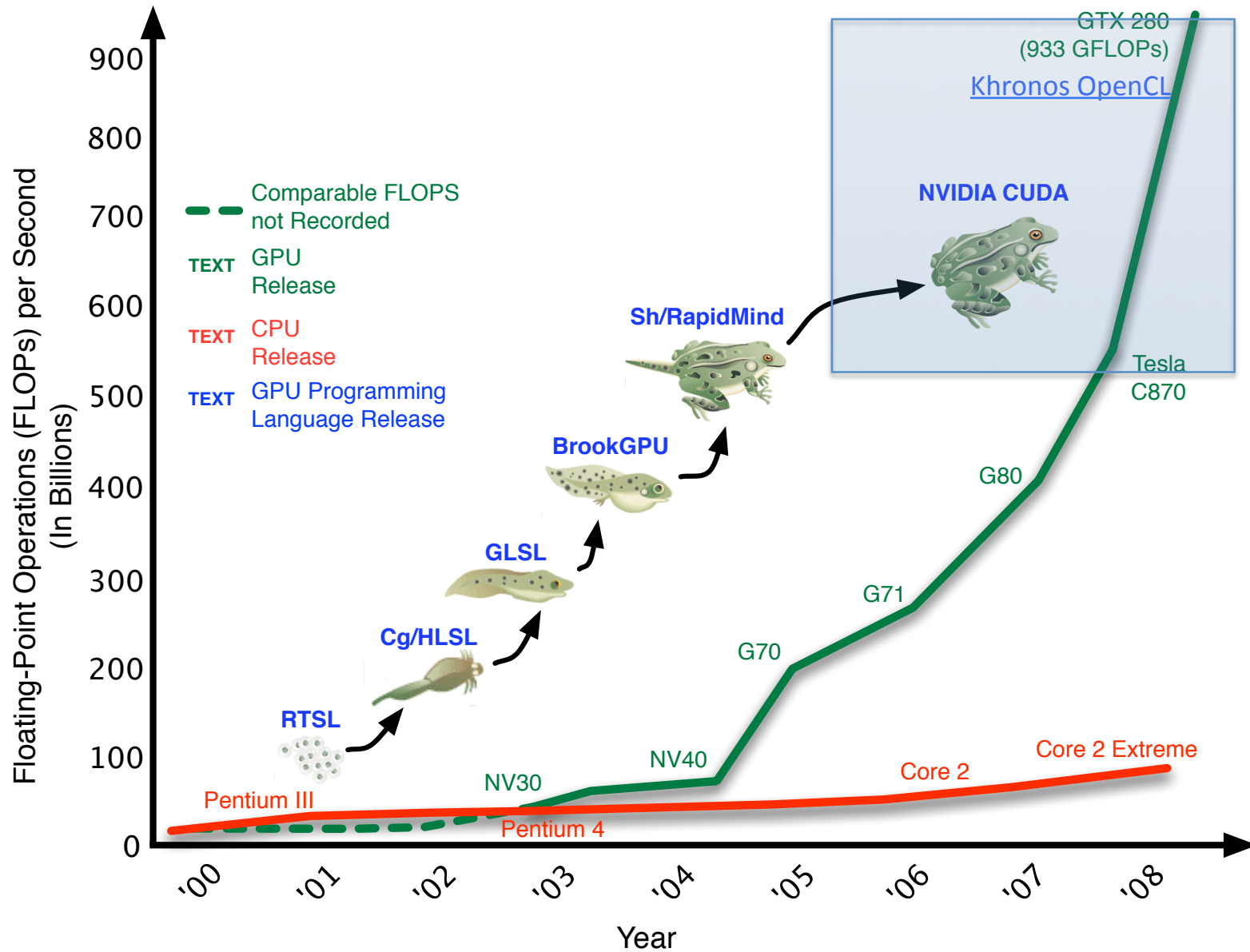
Theoretical GFLOP/s



Theoretical GB/s



Evolution GPU Languages Fermi, GTX480



TECHNICAL SPECS

Product	Tesla C870
Form Factor	ATX, 4.38" x 12.28"
# of Tesla GPUs	1
Total Dedicated Memory	1.5 GB GDDR3
Peak Flops	Over 500 gigaflops
Floating Point Precision	IEEE 754 single-precision floating point
Memory Interface	384-bit
Memory Bandwidth	76.8 GB/sec.
Max Power Consumption	170W
System Interface	PCI Express x16
Auxiliary Power Connectors	Yes (2)
Number of Slots	2
Thermal Solution	Active Fansink

Tesla 1060: Nov. 2008

Form Factor	10.5" x 4.376", Dual Slot
# of Tesla GPUs	1
# of Streaming Processor Cores	240
Frequency of processor cores	1.3GHz
Single Precision floating point performance (peak)	933
Double Precision floating point performance (peak)	78
Floating Point Precision	IEEE 754 single & double
Total Dedicated Memory	4GB GDDR3
Memory Speed	800MHz
Memory Interface	512-bit
Memory Bandwidth	102GB/sec
Max Power Consumption	200 W peak, 160 W typical
System Interface	PCIe x16
Auxiliary Power Connectors	6-pin & 8-pin
Thermal Solution	Active fan sink
Programming environment	CUDA

GTX 480 (graphics Fermi)

GPU Engine Specs:

CUDA Cores	480
Graphics Clock (MHz)	700 MHz
Processor Clock (MHz)	1401 MHz
Texture Fill Rate (billion/sec)	42

Memory Specs:

Memory Clock (MHz)	1848
Standard Memory Config	1536 MB GDDR5
Memory Interface Width	384-bit
Memory Bandwidth (GB/sec)	177.4

Feature Support:

	GTX 680	GTX 580	GTX 560 Ti	GTX 480
Stream Processors	1536	512	384	480
Texture Units	128	64	64	60
ROPs	32	48	32	48
Core Clock	1006MHz	772MHz	822MHz	700MHz
Shader Clock	N/A	1544MHz	1644MHz	1401MHz
Boost Clock	1058MHz	N/A	N/A	N/A
Memory Clock	6.008GHz GDDR5	4.008GHz GDDR5	4.008GHz GDDR5	3.696GHz GDDR5
Memory Bus Width	256-bit	384-bit	256-bit	384-bit
Frame Buffer	2GB	1.5GB	1GB	1.5GB
FP64	1/24 FP32	1/8 FP32	1/12 FP32	1/12 FP32
TDP	195W	244W	170W	250W
Transistor Count	3.5B	3B	1.95B	3B
Manufacturing Process	TSMC 28nm	TSMC 40nm	TSMC 40nm	TSMC 40nm
Launch Price	\$499	\$499	\$249	\$499

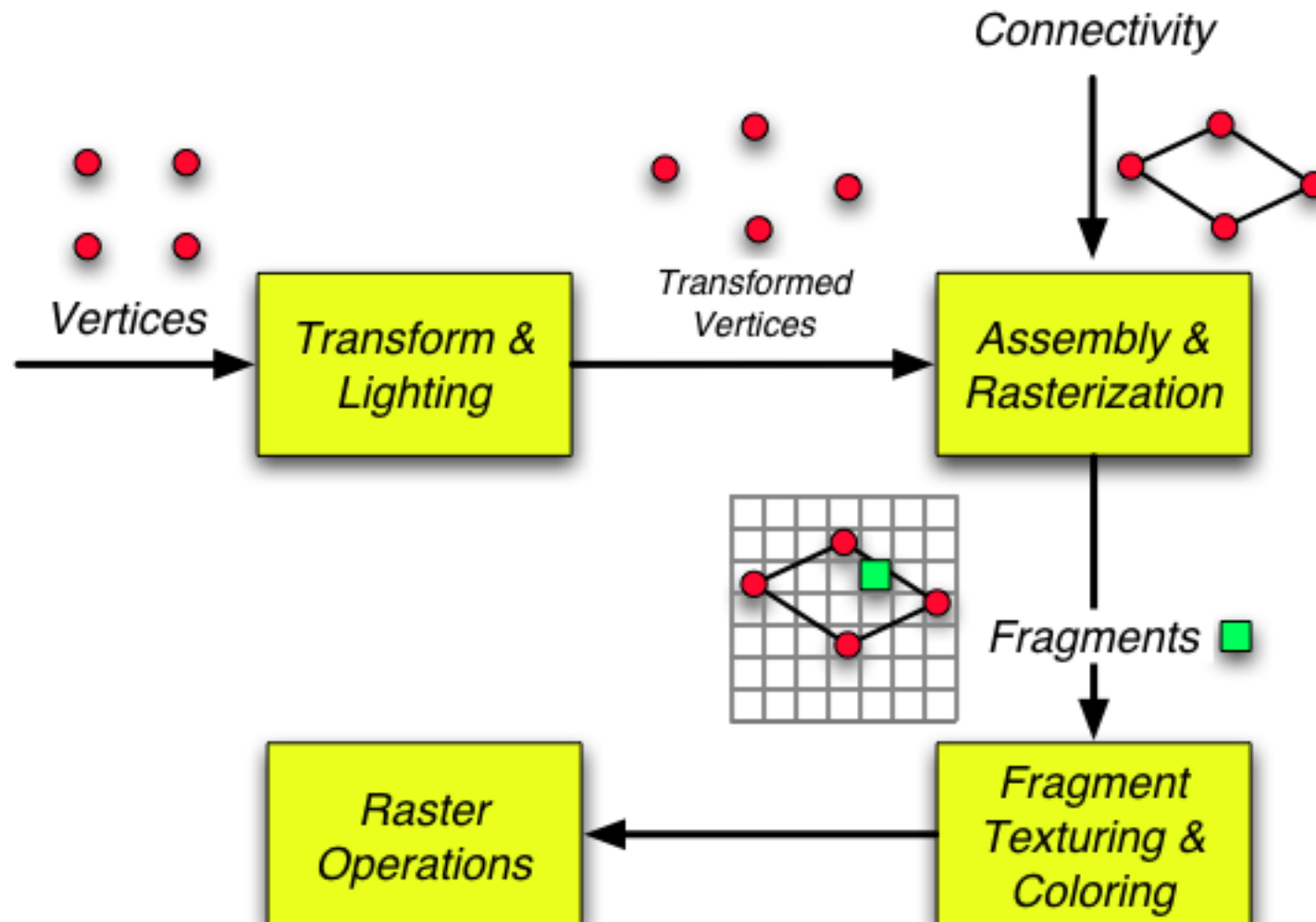
Kepler (2012)

- Nvidia has concentrated on power reduction rather than increase in computational speed
 - 3x performance per Watt
- http://www.nvidia.com/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf
- Thread spawning without returning to CPU
- A kernel can launch other kernels without CPU
- Multiple CPU cores can launch work on single GPU simultaneously
- Hyper-Q (performance boost for MPI)
- I have not yet studied the Kepler

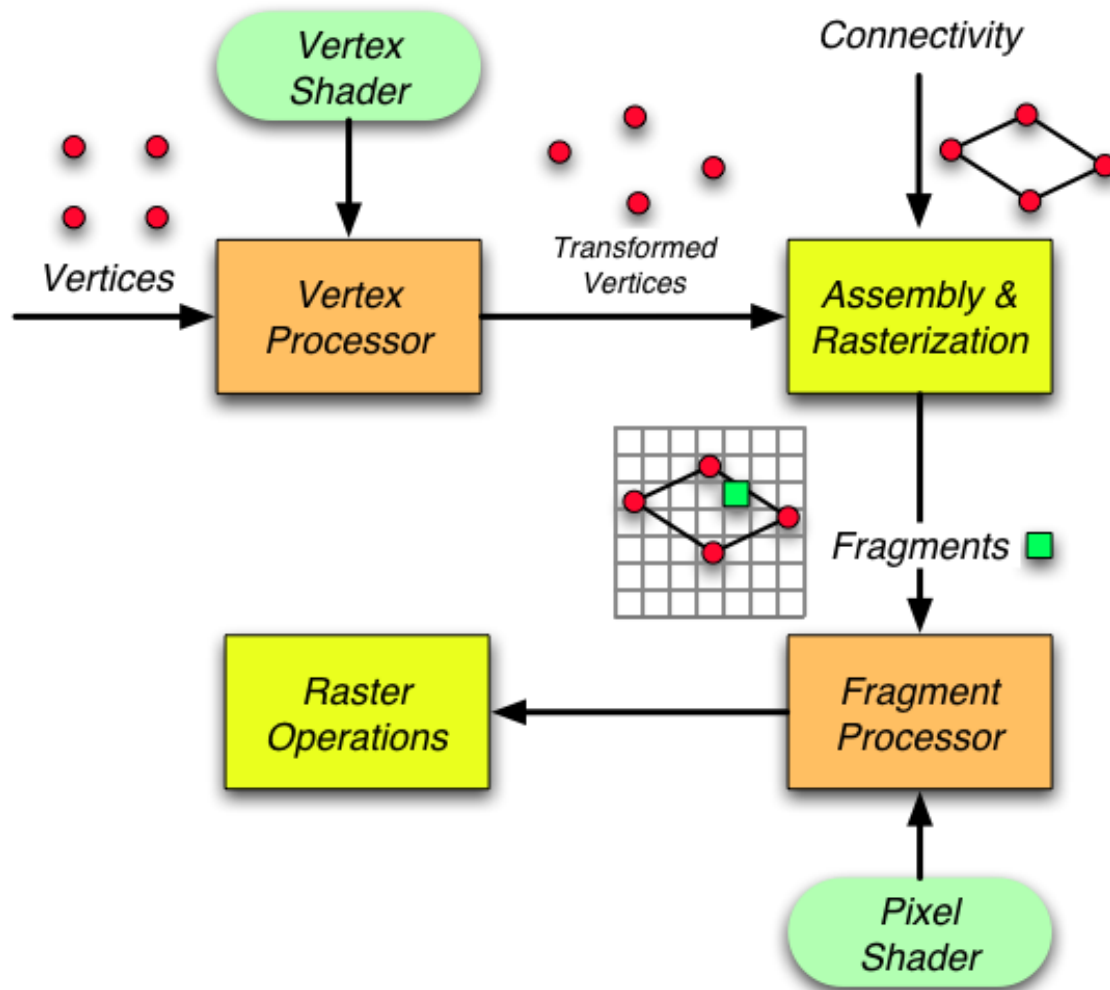
Graphic Card Evolution on PC

- **Increased graphics functionality is relegated to the graphics card**
 - < 1995 : fixed pipeline. Specialized workstations: Silicon Graphics (\$100k)
Amiga: first PC with graphics controlled by special card
 - 1995-1998: 2D acceleration, textures and z-buffer
 - 1998: multi-textures
 - 1999-2000: 3D transformations and lighting (Geforce 256)
 - 2001: first programmable vertex shaders
 - 2002-2003: first programmable pixel shaders, looping
 - 2004: Shader 3.0 + 64-bit color
 - 2005: Multi-GPUs, conditionals
 - 2007: Shader 4.0 (geometry shaders), Unified architecture (CUDA)
 - 2008: GPU clusters
 - 2009: OpenCL (Khronos)

Fixed Graphic Pipeline (- 2000)



Graphic Pipeline with Programmable Shaders (2000 – 2007)



Low level Programming

```
#Phong lighting
#compute half angle vector
ADD spec.rgb, view, lVec;
DP3 spec.a, spec, spec;
RSQ spec.a, spec.a;
MUL spec.rgb, spec, spec.a;

#compute specular intensisty
DP3_SAT spec.a, spec, tmp;
LG2 spec.a, spec.a;
MUL spec.a, spec.a, const.w;
EX2 spec.a, spec.a;

#compute diffuse illum
DP3_SAT dif, tmp, lVec;
ADD_SAT dif.rgb, dif, const;
```

Highlights

GLSL/HLSL/Cg

```
Void main() {  
    vec4 alpha = texture2DRect(maskBoundariesCond, gl_TexCoord[0].xy);  
  
    // Read the neighboring texture values  
    vec4 u = texture2DRect(initialsValues, gl_TexCoord[0].xy);  
    vec4 u_right = texture2DRect(initialsValues, gl_TexCoord[0].xy + vec2(h,0.0));  
    vec4 u_left = texture2DRect(initialsValues, gl_TexCoord[0].xy - vec2(h,0.0));  
    vec4 u_up = texture2DRect(initialsValues, gl_TexCoord[0].xy + vec2(0.0,h));  
    vec4 u_bottom = texture2DRect(initialsValues, gl_TexCoord[0].xy - vec2(0.0,h));  
    vec4 result = u + dt*((u_right + u_left + u_up + u_bottom) - 4.*u) - f_xyt(u);  
    float normx = 0.5*(u_right - u_left).x;  
    float normy = 0.5*(u_up - u_bottom).x;  
    vec3 norm = vec3(normx, normy, 1./float(tex_size));  
    gl_FragColor = vec4(result.x, norm); // return |grad(u)| in vec[1], vec[2]  
}
```

CUDA: Transpose Naïve

C-like programming

```
// This naive transpose kernel suffers from completely non-coalesced writes.  
// It can be up to 10x slower than the kernel on following page for large  
  matrices.  
__global__ void transpose_naive(float *odata, float* idata, int width, int height)  
{  
    unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;  
    unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;  
  
    if (xIndex < width && yIndex < height)  
    {  
        unsigned int index_in  = xIndex + width * yIndex;  
        unsigned int index_out = yIndex + height * xIndex;  
        odata[index_out] = idata[index_in];  
    }  
}
```

CUDA: Transpose (more efficient)

```
__global__ void transpose(float *odata, float *idata, int width, int height) {
    __shared__ float block[(BLOCK_DIM+1)*BLOCK_DIM];
    unsigned int xBlock = __mul24(blockDim.x, blockIdx.x);
    unsigned int yBlock = __mul24(blockDim.y, blockIdx.y);
    unsigned int xIndex = xBlock + threadIdx.x;
    unsigned int yIndex = yBlock + threadIdx.y;
    unsigned int index_out, index_transpose;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in = __mul24(width, yIndex) + xIndex;
        unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM+1) + threadIdx.x;

        block[index_block] = idata[index_in];
        index_transpose = __mul24(threadIdx.x, BLOCK_DIM+1) + threadIdx.y;
        index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
    }
    __syncthreads();

    if (xIndex < width && yIndex < height) {
        odata[index_out] = block[index_transpose];
    }
}
```


Parallelism

Task parallelism

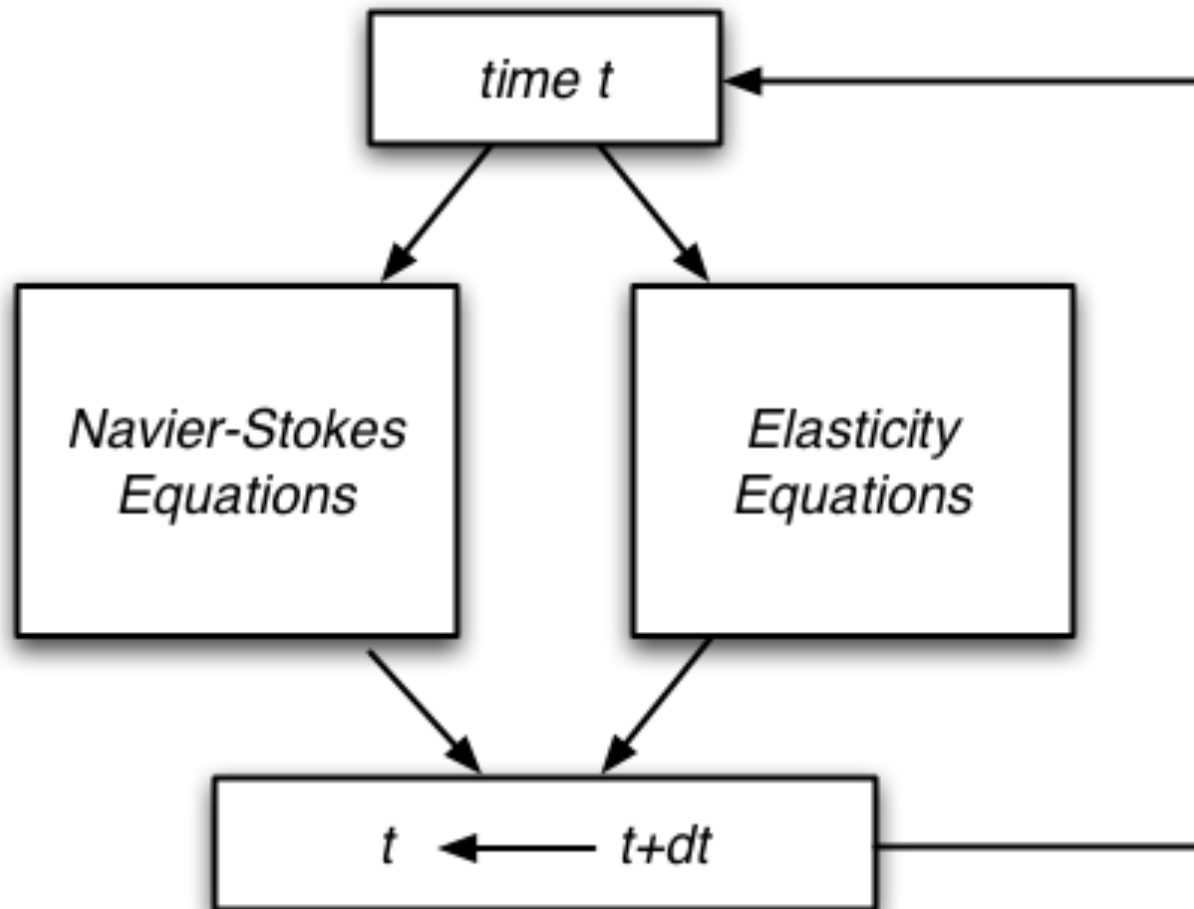
Data parallelism

SIMD

MIMD

Stream

Task Parallelism



Data Parallelism

Add 2 matrices
 $C(i,j) = A(i,j) + B(i,j)$

$$C(1,1) = A(1,1) + B(1,1)$$

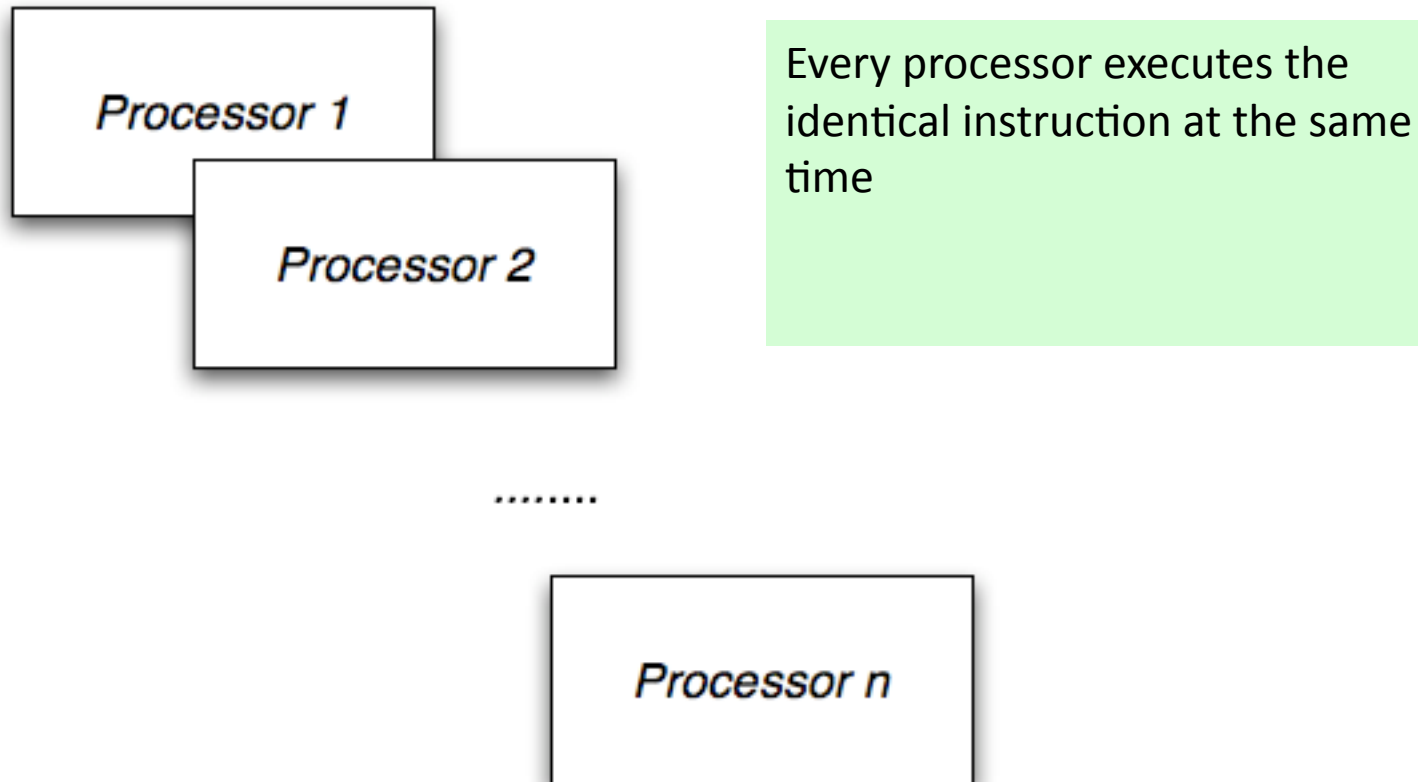
$$C(2,1) = A(2,1) + B(2,1)$$

$$C(3,1) = A(3,1) + B(3,1)$$

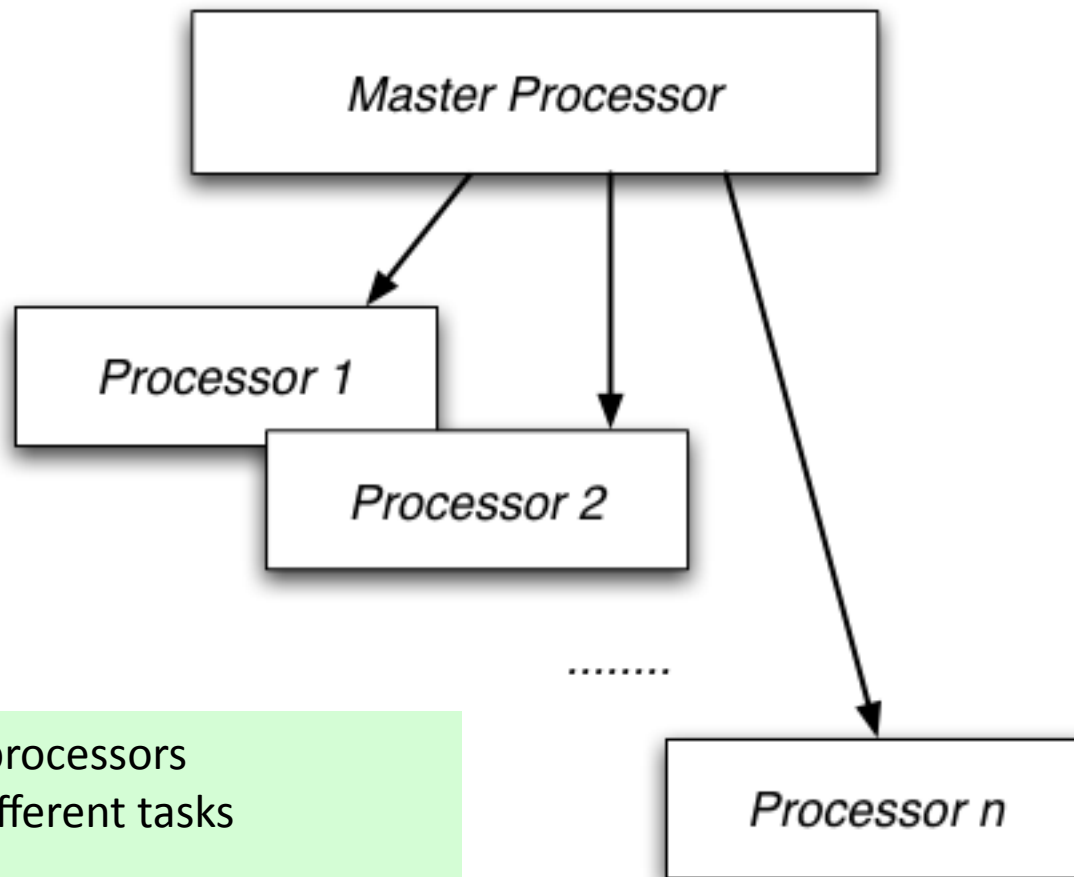
.....

$$C(n,m) = A(n,m) + B(n,m)$$

Single Instruction Multiple Data

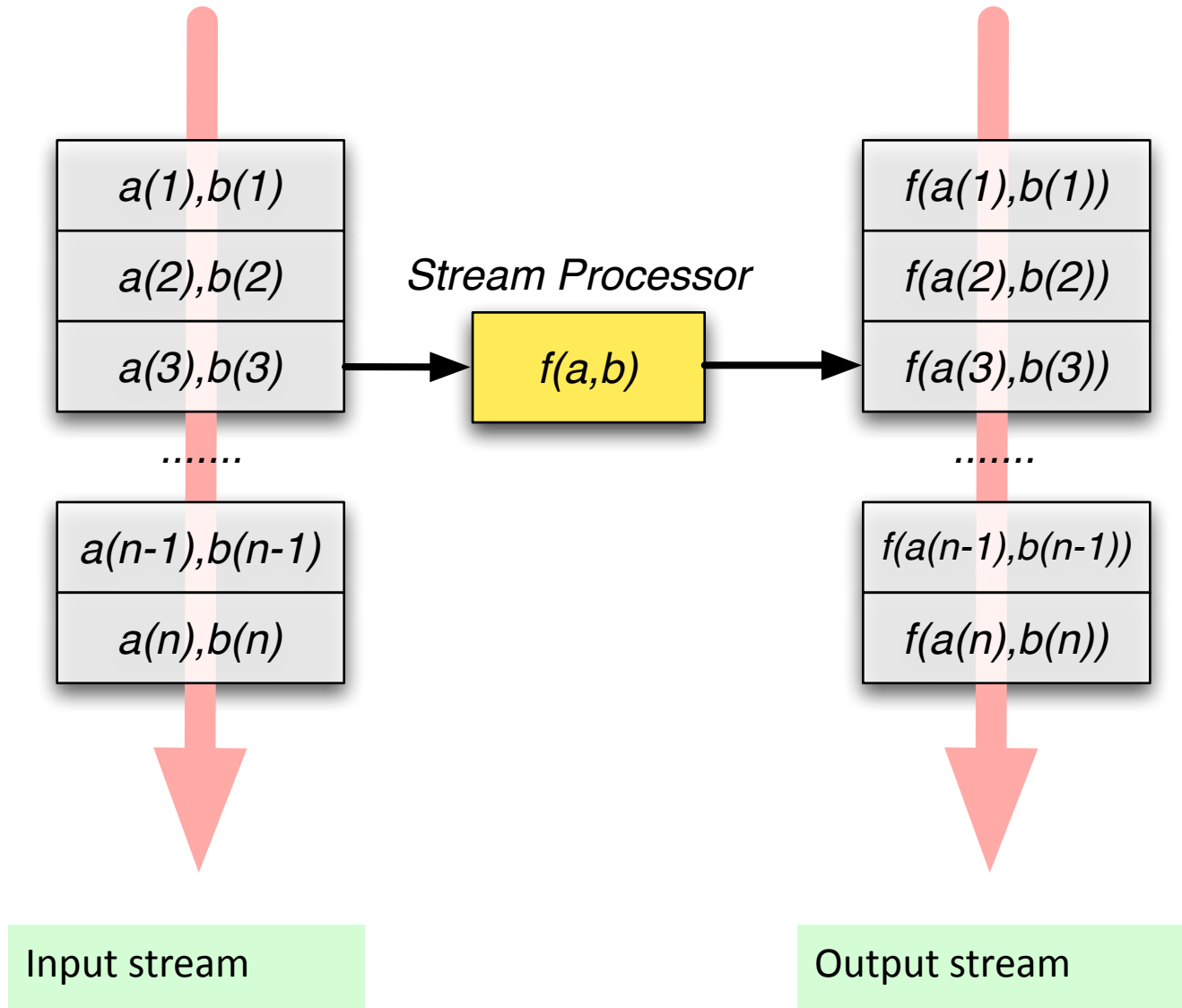


Multiple Instruction Multiple Data

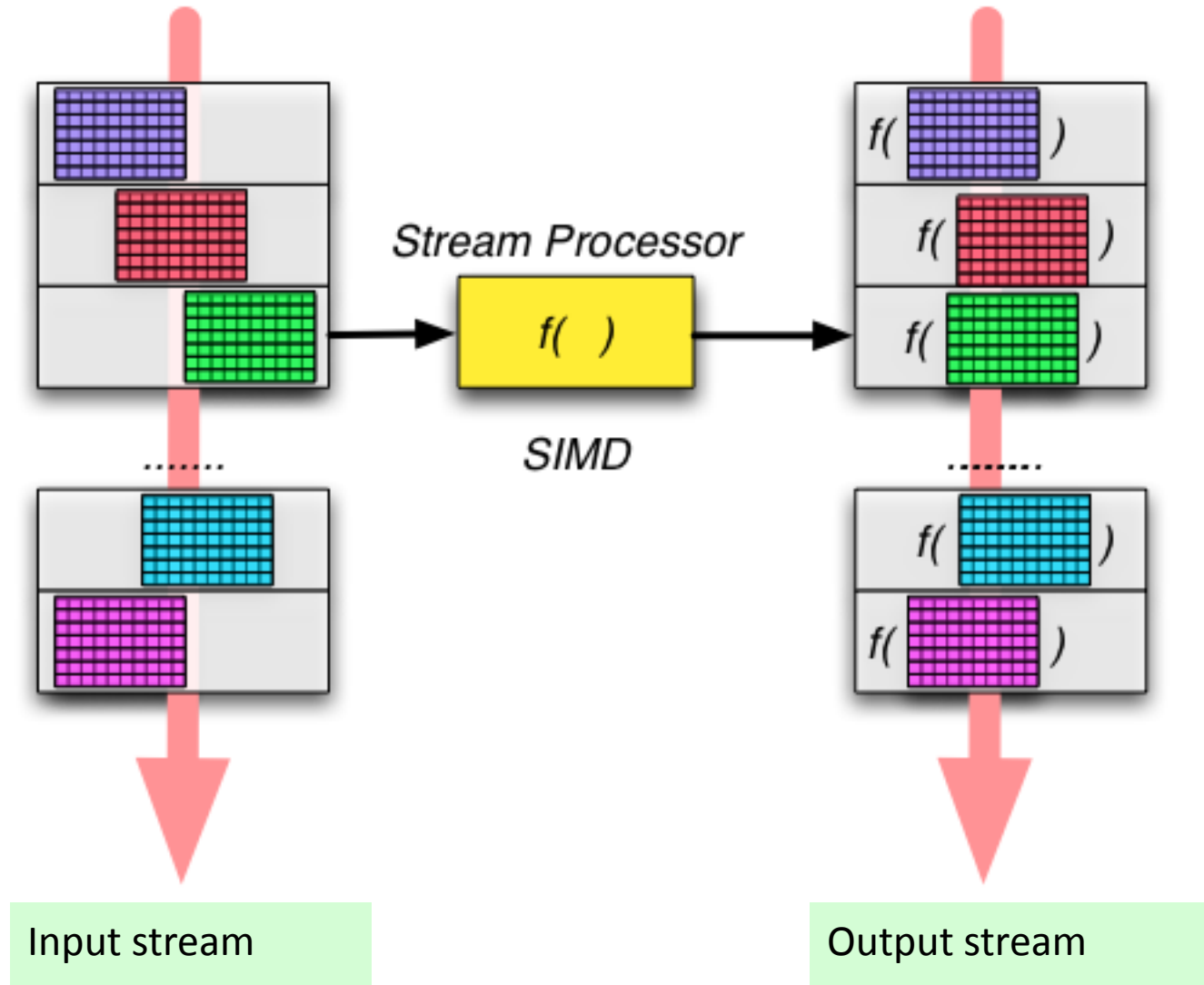


Different processors
execute different tasks

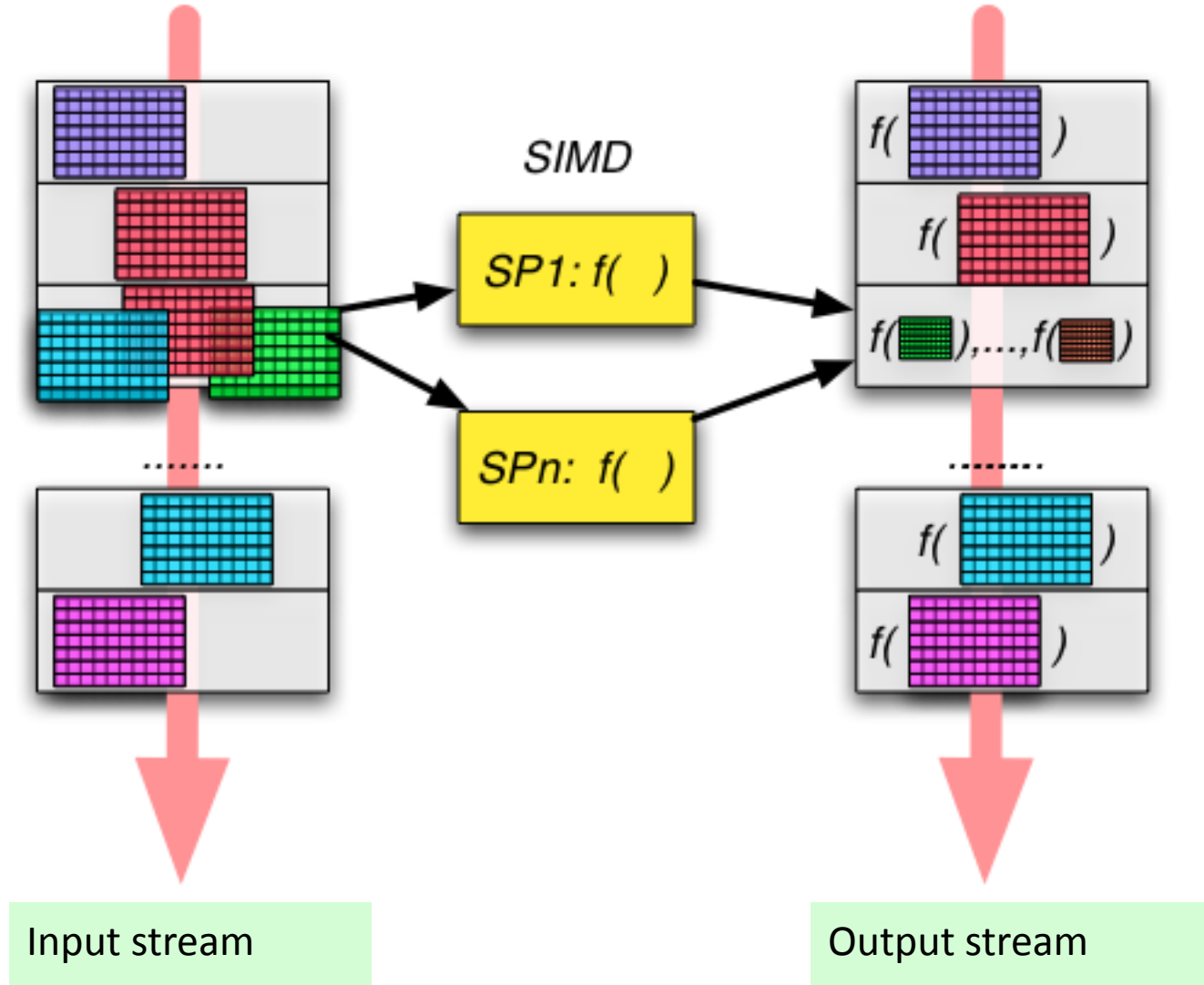
Stream Processing



Stream Processing block of threads



Stream Processing block of threads



Computer
Unified
Data
Architecture

Based on the GTX480 (2010)

Acronym is no longer used

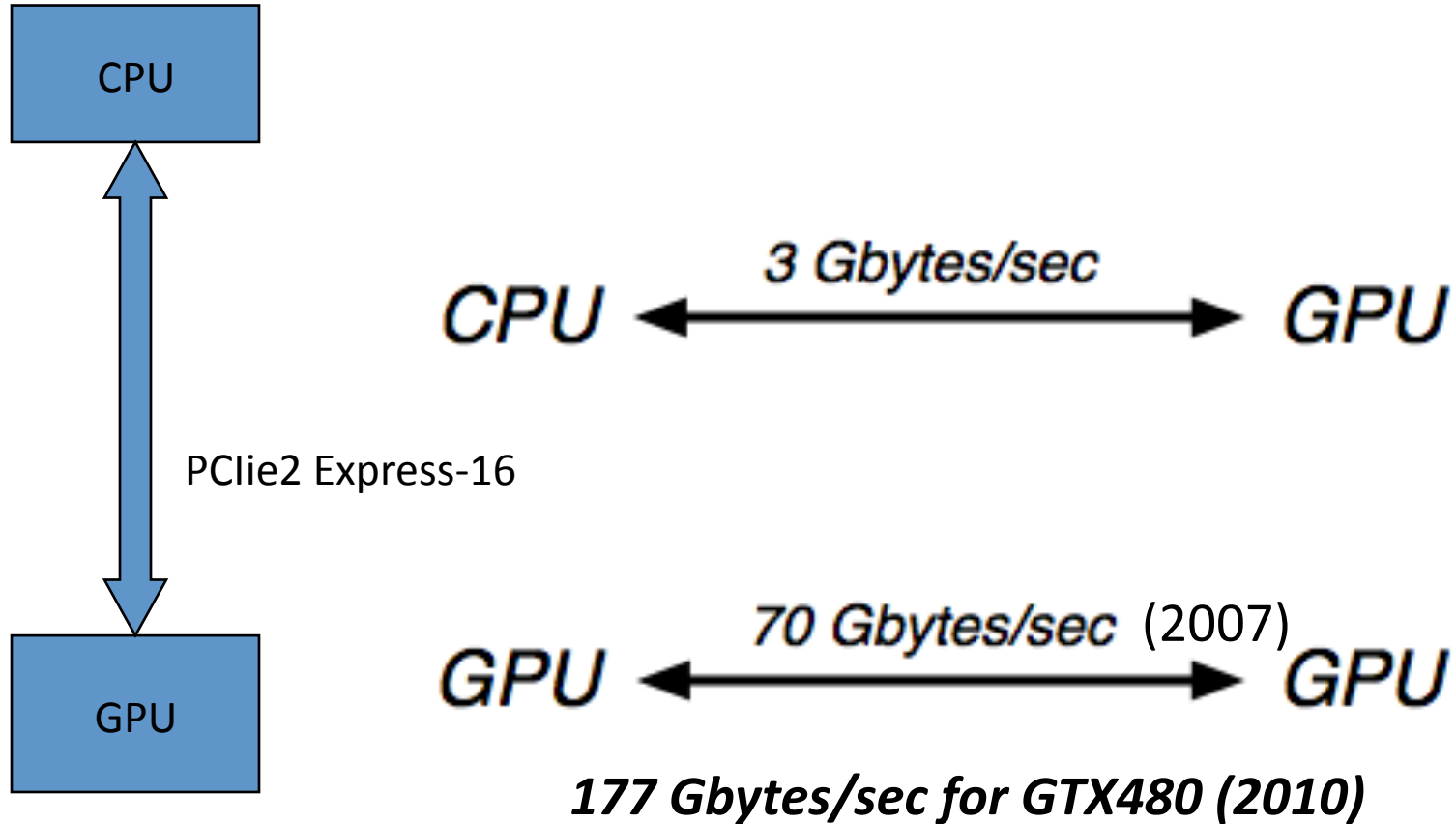
CUDA

- Compute Unified Device Architecture
- Stream Processor
- Remove graphics aspect of GPUs
- Geared towards scientific programming
- Can do graphics programming by moving data from an array to a OpenGL framebuffer and then using standard GPU programming

OpenCL

- A GPU-based language that
 - Leverages OpenGL (Graphics language)
 - Is portable across multiple vendors
 - Works on various graphics devices
 - Nvidia, ATI, multi-core chips, etc.
- Many similarities with CUDA
 - Less general
 - Less powerful
 - More portable
 - In early stages

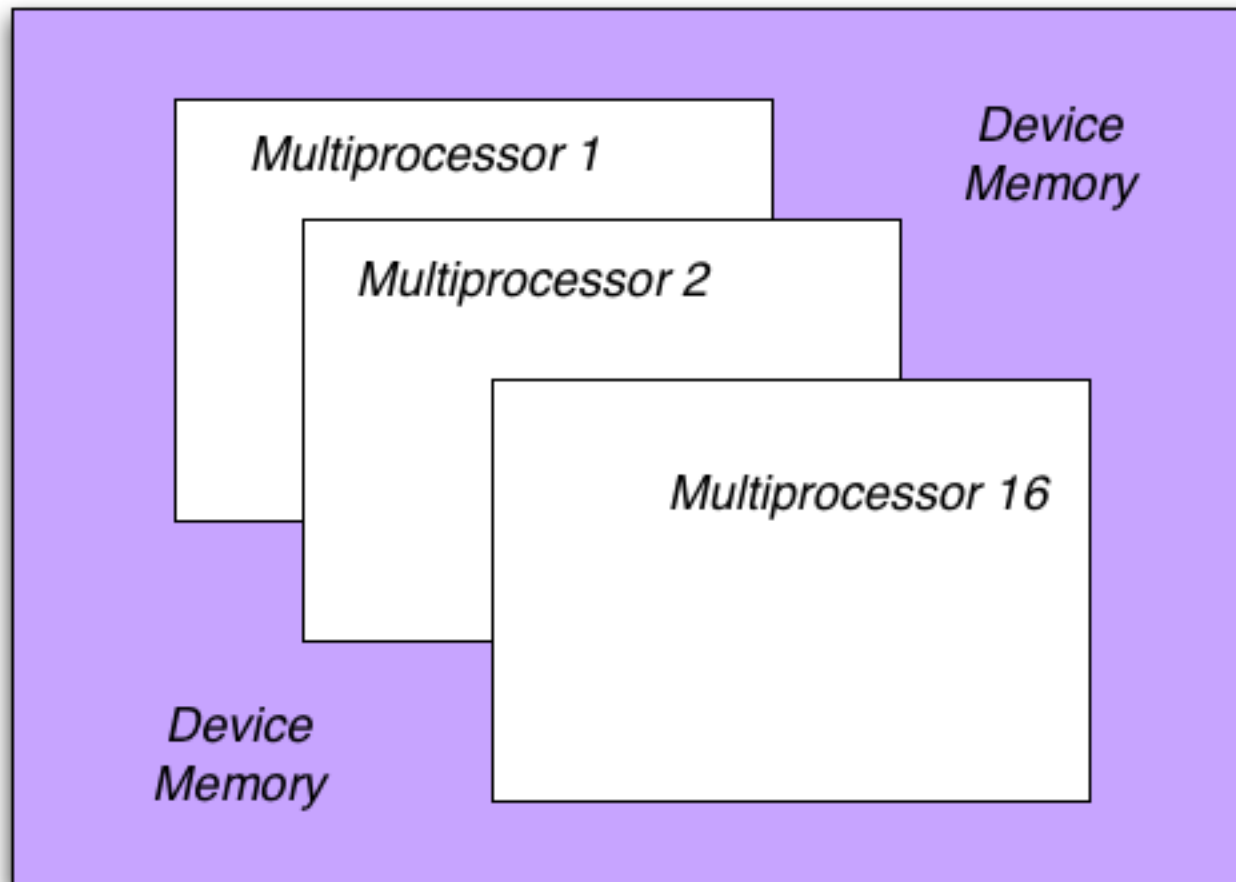
Memory Bandwidth



PCIe3 (2010), 5 Gbyte/sec

Increased disparity between speeds

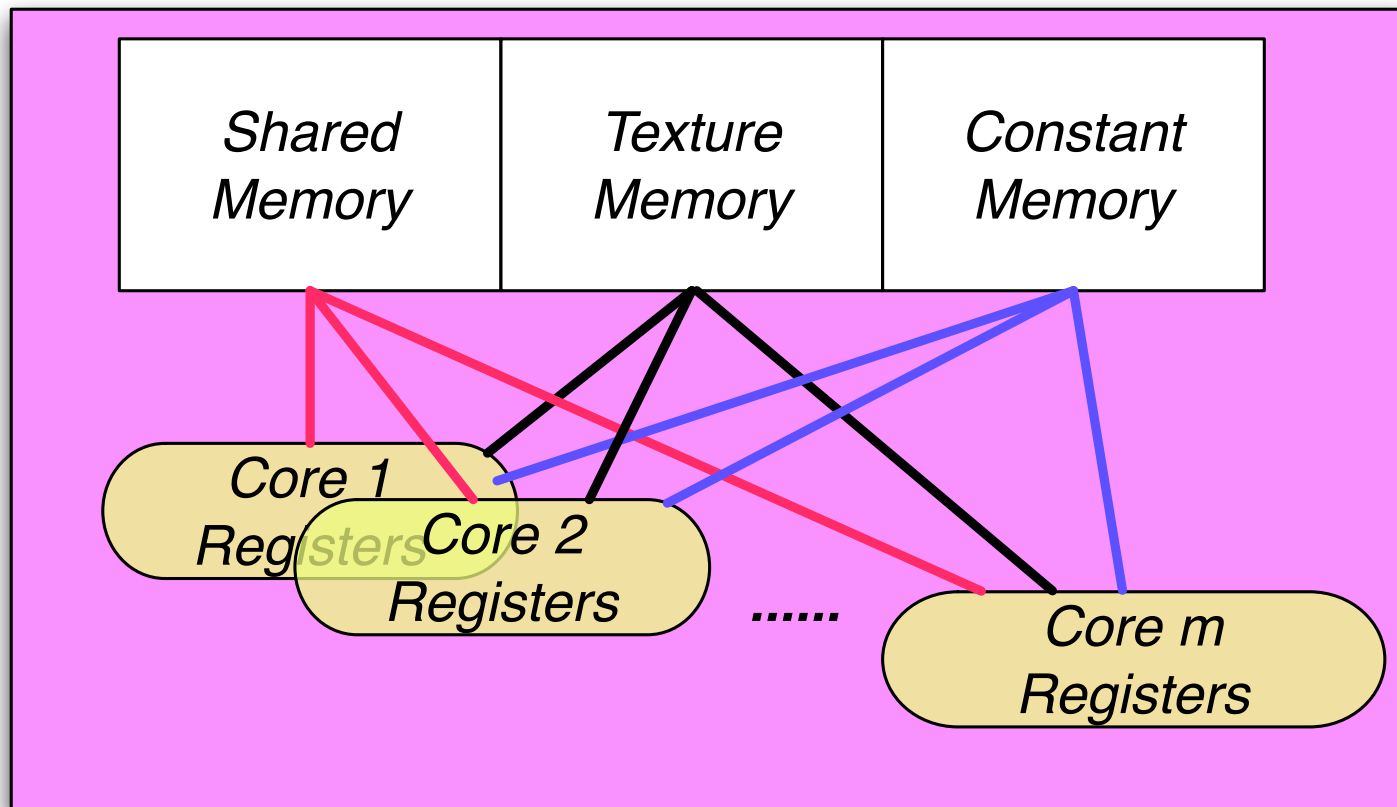
Several Streaming Multiprocessors
per GPU (280GTX has 24 MP)
480GTX has 15 MP

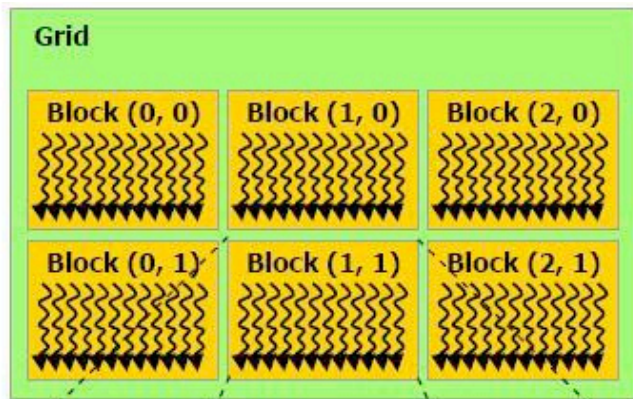


Streaming Multiprocessor

Multiple stream processors

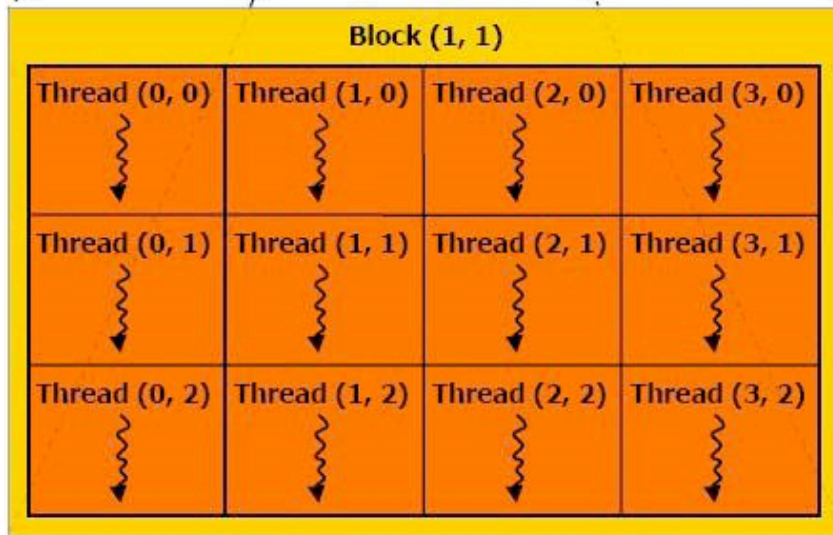
GTX480: 32 cores per SM





Input to GPU

Grid of Blocks



Single Stream Processor

Block of Threads

= a grid of blocks

Thread



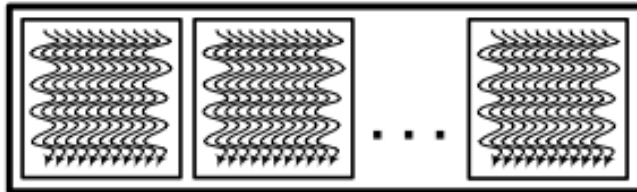
per-Thread Private Local Memory

Thread Block



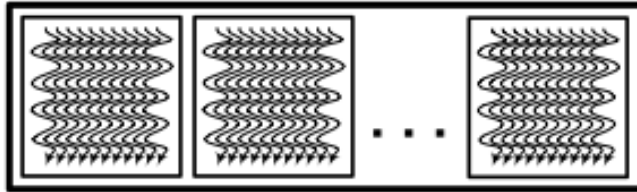
per-Block Shared Memory

Grid 0



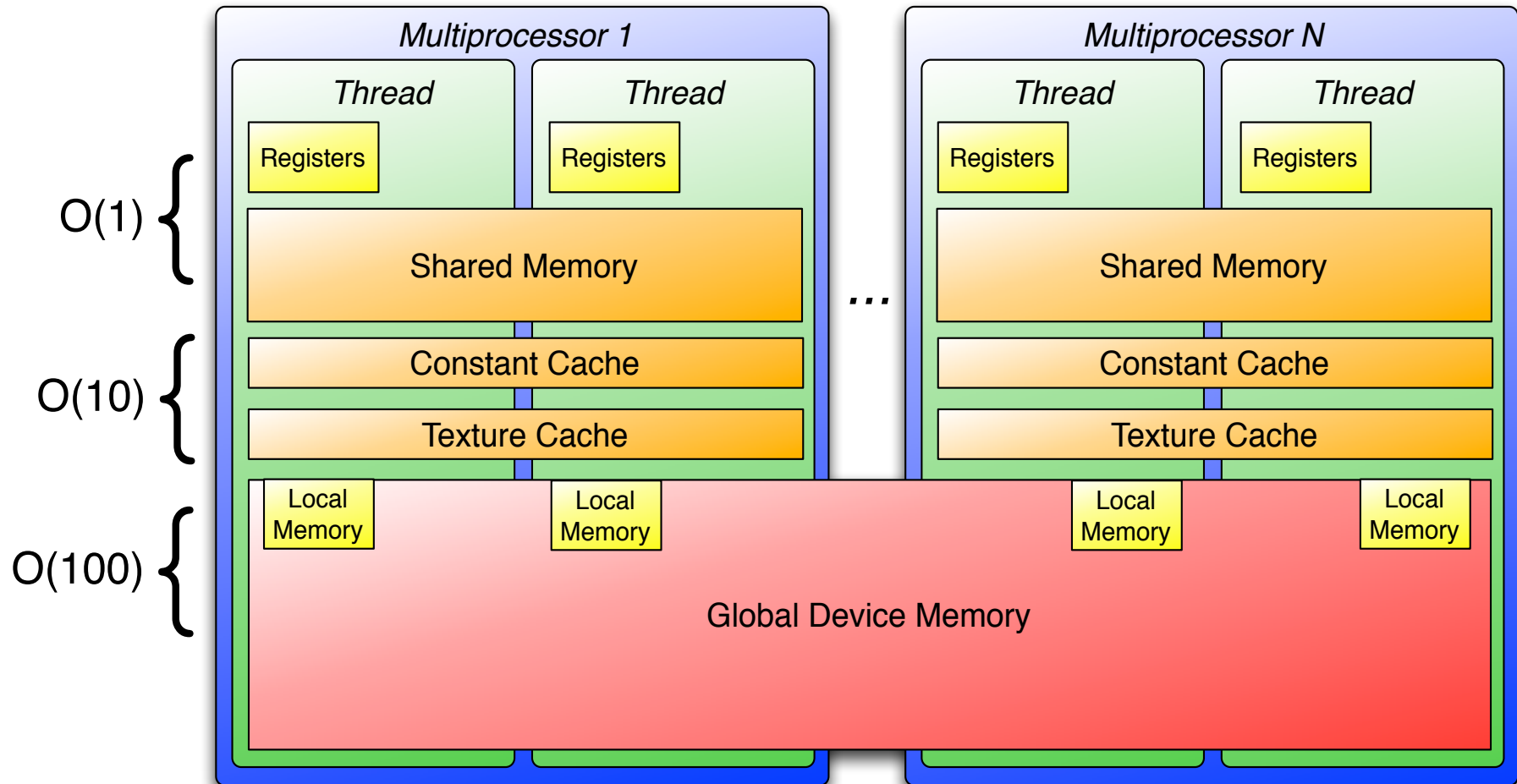
per-Application Context Global Memory

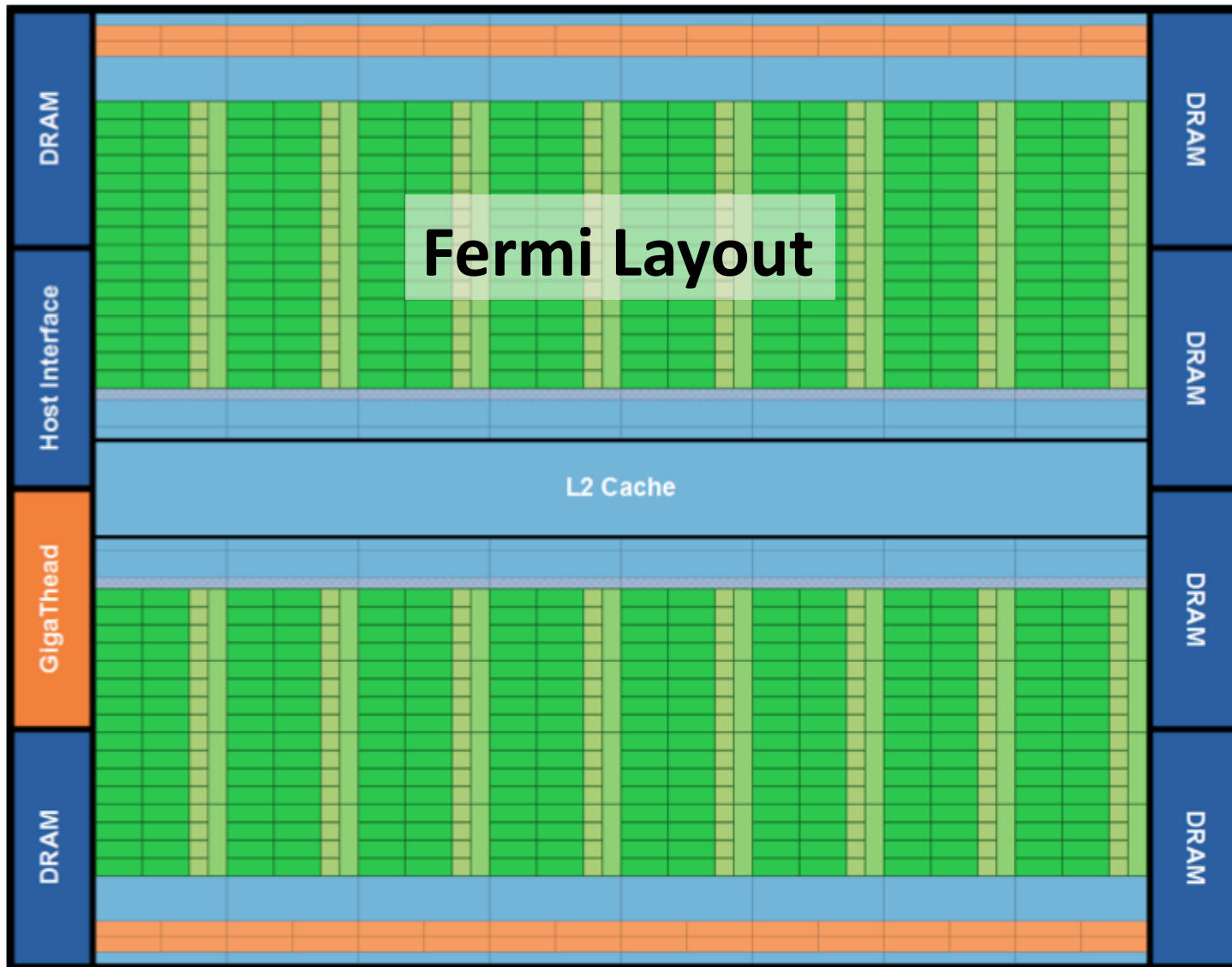
Grid 1



Anatomy of Unified Architecture

GEForce 8x





Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical rectangular strip that contain an orange portion (scheduler and dispatch), a green portion (execution units), and light blue portions (register file and L1 cache).

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GT 325M			
GeForce 9700M GT, GT 240M, GT 230M	1.1	6	48
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	1.1	4	32
GeForce 210, 310M, 305M	1.2	2	16
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU, G210M, G110M	1.1	2	16
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G, G105M	1.1	1	8
Tesla C2050	2.0	14	448
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240
Tesla S870	1.0	4x16	4x128
Tesla D870	1.0	2x16	2x128
Tesla C870	1.0	16	128

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 480	2.0	15	480
GeForce GTX 470	2.0	14	448
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280, GTX 275	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 260M, 9800M GTX	1.1	14	112
GeForce GT 240, GTS 360M, GTS 350M	1.2	12	96
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTS 260M, GTS 250M, 9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce GT 335M	1.2	9	72
GeForce 9600 GT, 8800M GTS,	1.1	8	64

Technical Specifications	1.0	1.1	1.2	1.3	2.0
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24	32		48	
Maximum number of resident threads per multiprocessor	768	1024		1536	
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB		
Maximum width for a 1D texture or surface reference bound to a CUDA array	8192		32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}		
Maximum width and height for a 2D texture reference bound to linear memory or for a 2D texture or surface reference bound to a CUDA array	65536 x 32768		65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048		
Maximum number of textures that can be bound to a kernel	128		
Maximum number of surfaces that can be bound to a kernel	8		
Maximum number of instructions per kernel	2 million		

CUDA on GeForce 8800

- # Multiprocessors: depends on card
- Each Multiprocessor: 8 stream processors
 - (→ 8 concurrent blocks)
- Warp size: **32** = 2 half-warps
- Max # threads per block: 512, # warps per block: 16
- Registers per MP: 8192 (*faster access*)
- Shared memory: 16,000 bytes per MP (*fast access*)
- Constant memory: 64,000 bytes + 8 kB cache per multiprocessor (*fast access*)
- Max concurrent blocks that can run concurrently on MP: 8
- Max # warps that can run concurrently on MP: **24**
- Max # threads that can run concurrently on MP: $768 = \mathbf{24} * \mathbf{32}$
- Max kernel size: 2 million instructions

CUDA on GTX480

- # Multiprocessors: 15
- Each Multiprocessor: 32 cores
- Warp size: 32
- Max # threads per block: 512, # warps per block: 16
- Registers per MP: 8192 (faster access)
- Shared memory: 48,000 bytes per MP (fast access)
- Constant memory: 64,000 bytes + 8 kB cache per multiprocessor (fast access)
- Max concurrent blocks that can run concurrently on MP: ??
- Max # warps that can run concurrently on MP: ??
- Max # threads that can run concurrently on MP: 1500
- Max kernel size: 2 million instructions

Programming CUDA

- **Easy to program, but there are multiple objectives (2007):**
 - Maximize number of concurrent running blocks (> 2-3 per MP)
 - Maximize number of concurrent running threads
 - 768 threads/MP => 96 threads/block if all processors are running
 - Optimum block size: $32 \times 16 = 512$
 - Shared memory is shared among the threads of a single block
 - The more blocks are running concurrently on a single MP, the less shared memory per block
 - # threads/block should be multiple of warp size
 - Keep enough registers per thread

Programming in CUDA

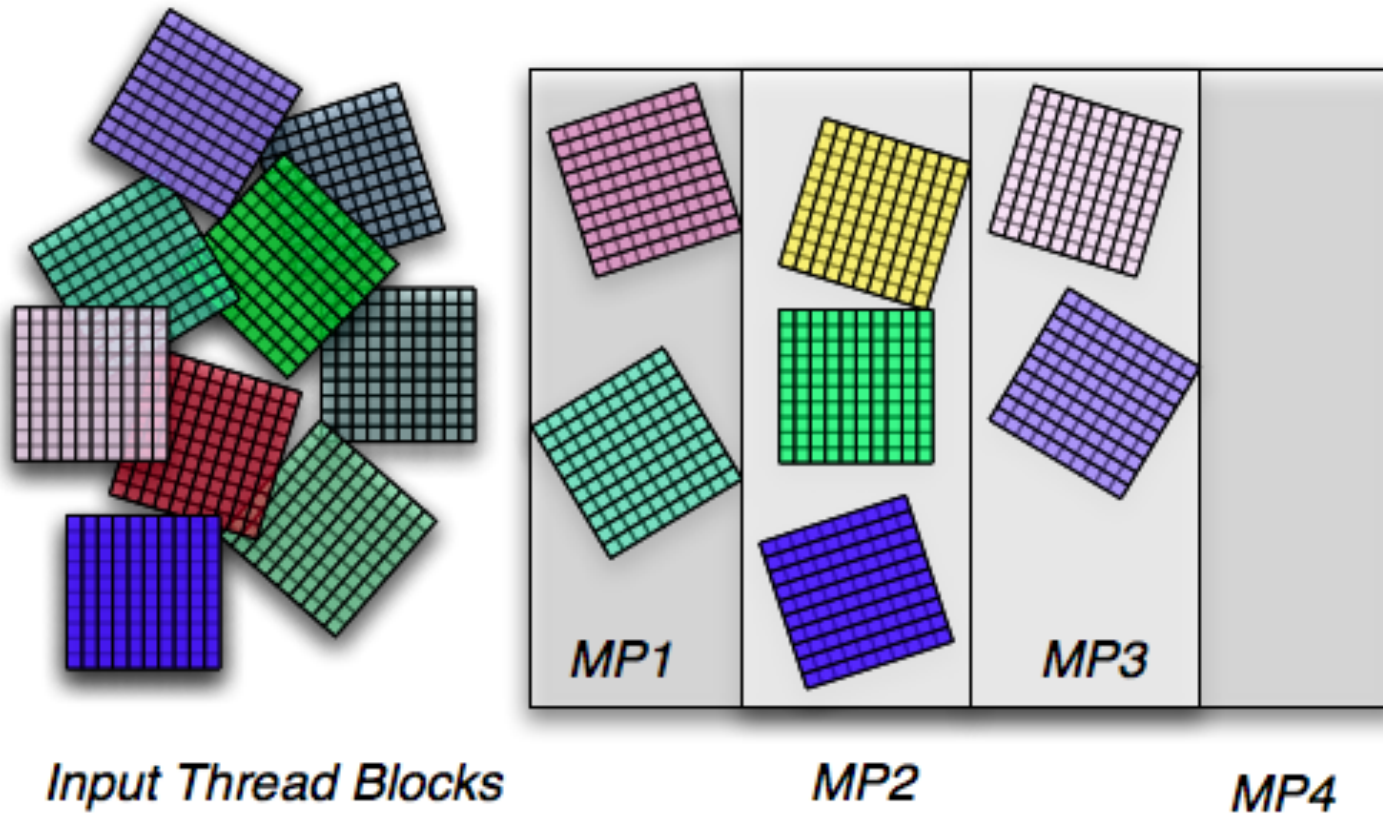
- **Lots of room for code optimization**
 - **More recent ideas:**
 - Maximize register usage
 - Can achieve close to peak performance with subset of threads
 - Maximize number of active threads
 - More important than active threads:
 - Ensure that all arithmetic units are full
 - If ALUs are 100% active, even with inactive threads, performance can no longer be enhanced

Programming CUDA

*Each multiprocessor has a Single Instruction,
Multiple Data architecture (SIMD)*

At any given clock cycle, each processor of each multiprocessor executes the same instruction, but operates on different data.

Treatment of input blocks



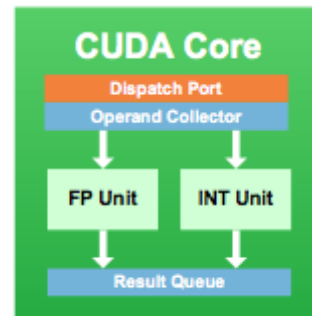
Efficient CUDA Programming

- Keep everything on the GPU
- Minimize communication GPU \leftrightarrow CPU
- Minimize transfers
 - Device Memory \leftrightarrow GPU registers
- Minimize number registers to use all threads
- Minimize incoherent reads from Device Memory
- Minimize bank conflicts from in shared memory
- Maximize use of shared memory
- Maximize number of blocks running concurrently
- **Conflicting requirements!**

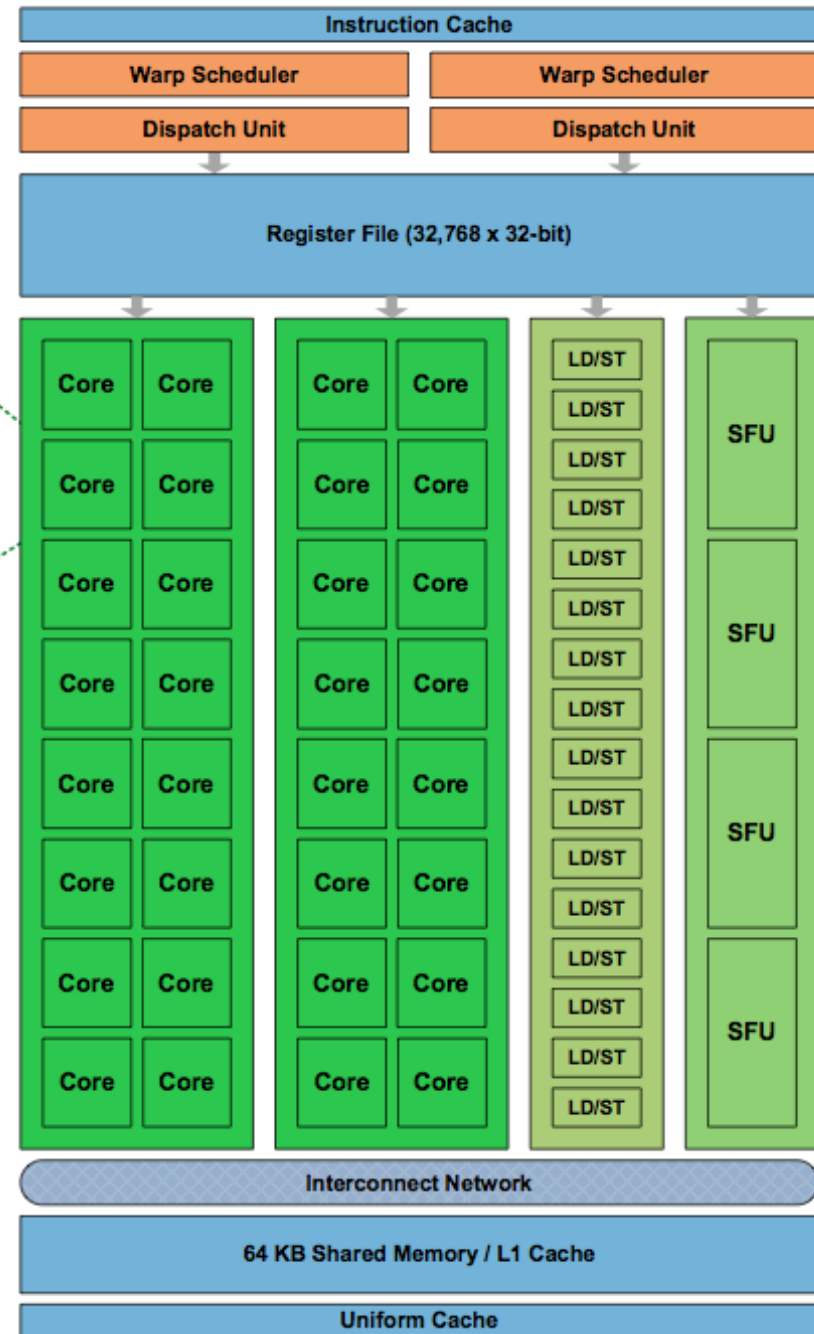
Efficient CUDA Programming

- Alternatives to maximizing number of threads
 - Maximize use of a single thread
 - Overlay I/O and arithmetic
- Need to study architecture of single Stream Multiprocessor in more detail
 - How are warps handled
 - Warp scheduling
 - How to achieve high performance on the Fermi?

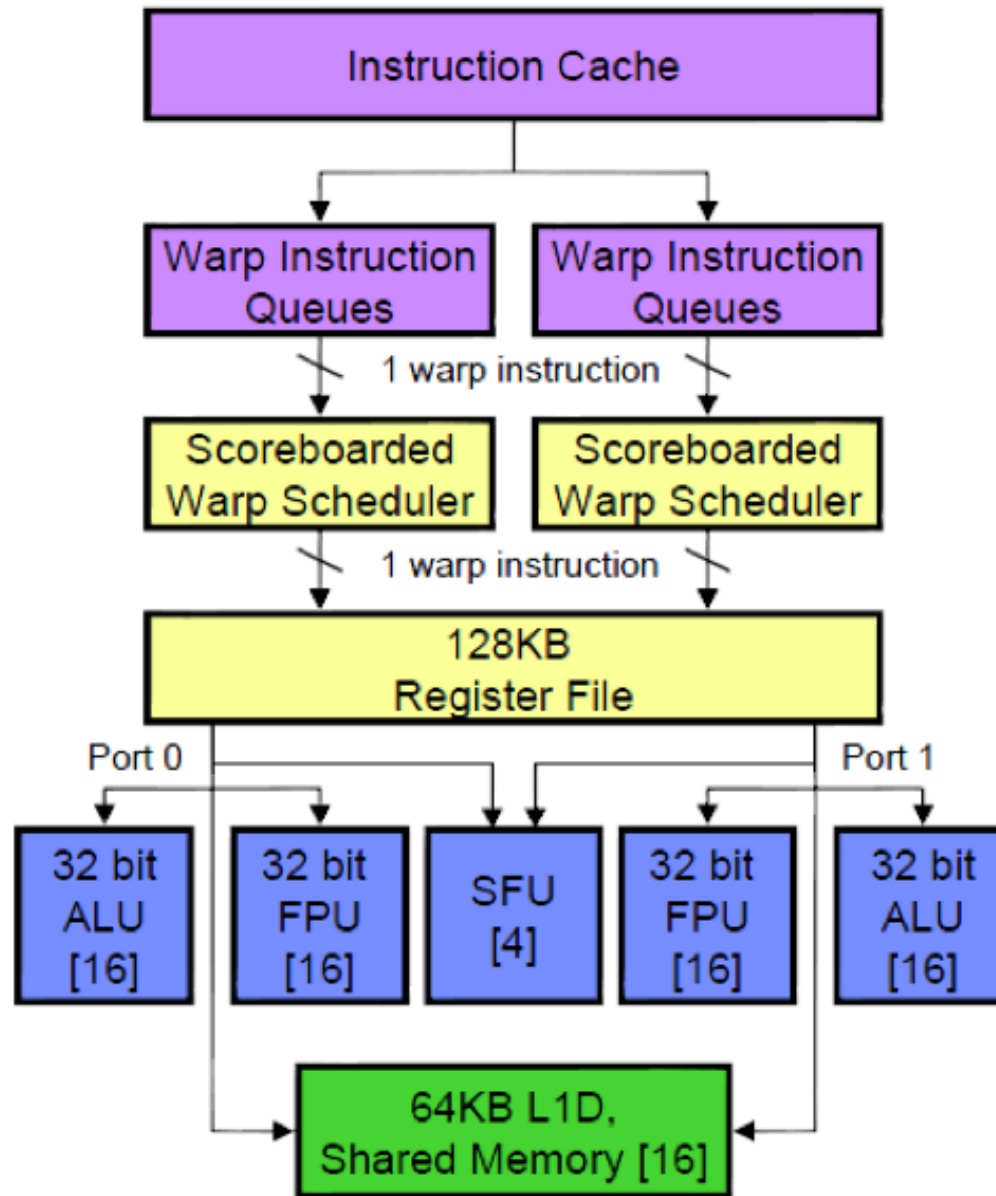
Single SM



32 cores
2 warp schedulers
2 dispatch units
4 SFU (special function)
16 LD/ST units
32 FP/INT units



Fermi Core

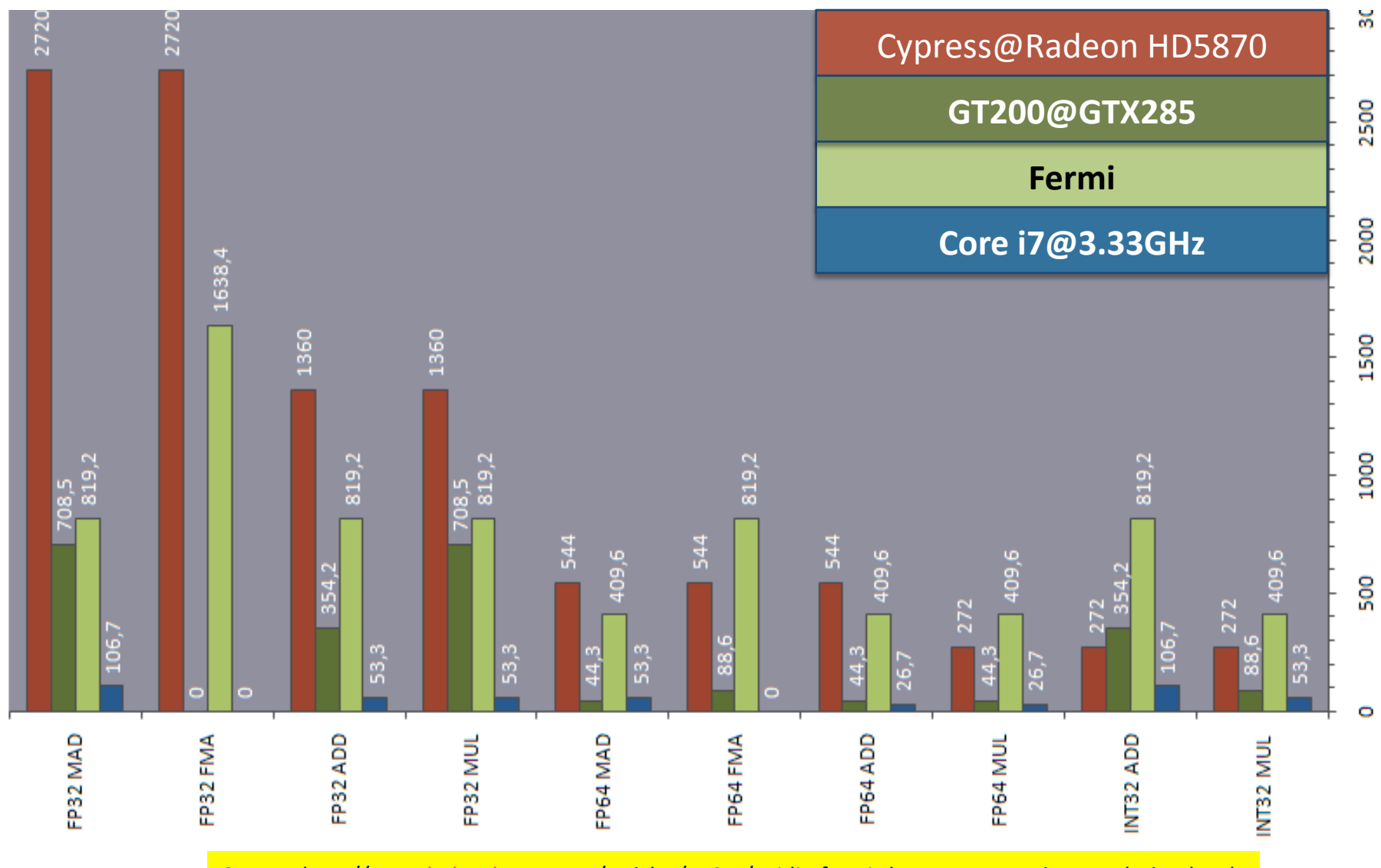


Single Multiprocessor

32 cores

- Cores divided into TWO execution blocks of 16 cores each
- Two ADDITIONAL execution blocks:
 - 16 load/store units
 - Four special function units (sin/cos/exp/log)
- At each cycle (clock), dispatch 32 instructions
 - From one or two warps to these blocks
 - It takes two cycles to issue instructions for a single warp (I/O and FPU) and 8 cycles for the SFU

Arithmetic Throughput



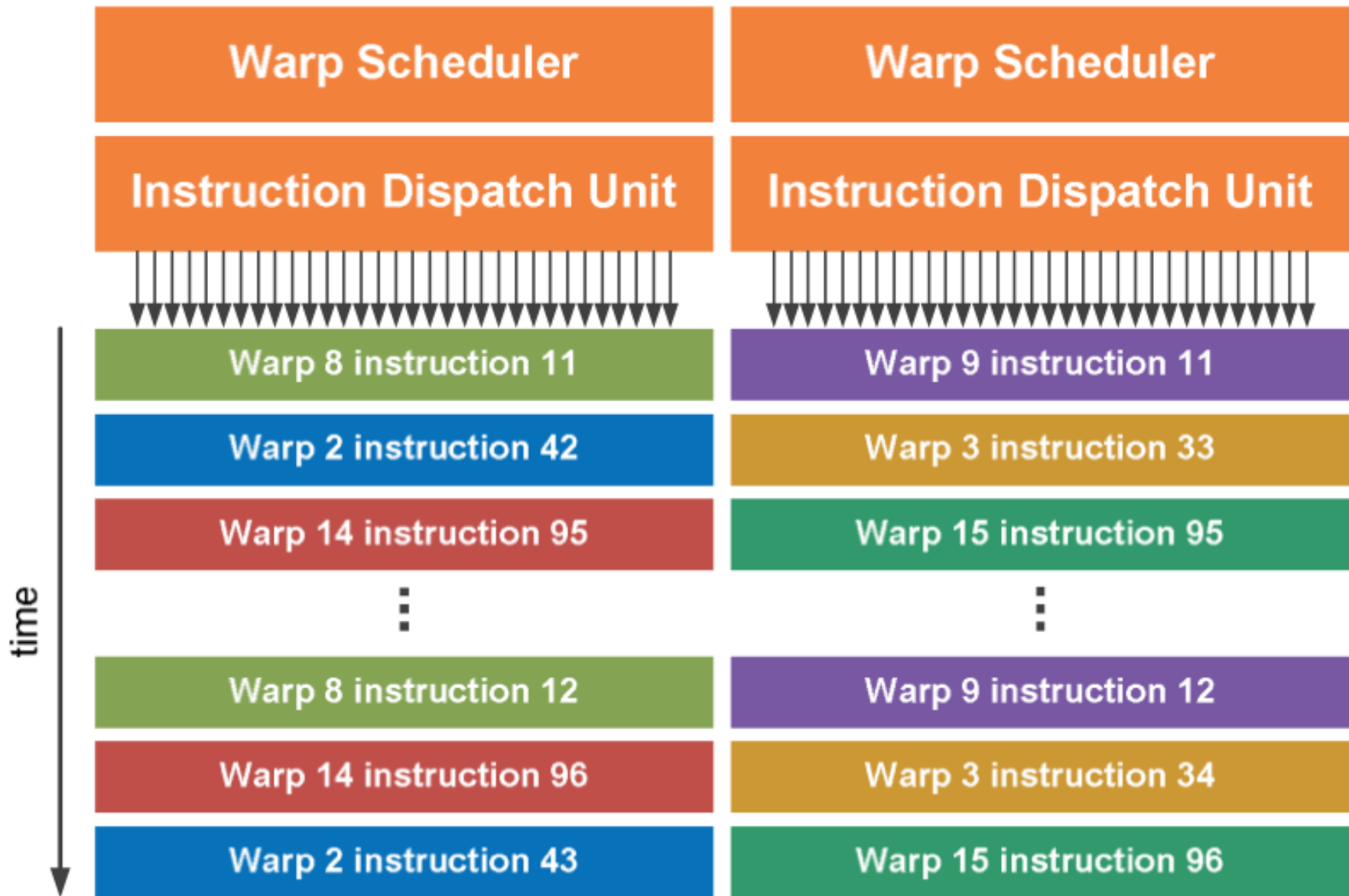
Source: <http://www.behardware.com/articles/772-7/nvidia-fermi-the-gpu-computing-revolution.html>

GTX480: Peak Flops

- Each cycle, single core
 - Initiate 16 flops (floating point operations)
 - Single flop = multiple, add, or multiply/add (MAD)
 - MAD: $a*x+b$
 - 32 cores: $32*16=512$ flops/cycle
 - Clock: 1.4 Ghz → 700 Gflop/sec
 - Theoretical max: MAD = 2 ops → 1.4 Tflop/sec

GTX480: peak memory throughput

- Each cycle, single SM
 - 16 x 32-bit Load/Store instructions
- Each cycle, entire GPU
 - $16 * 15 = 240$ Load/Store instructions
- Each second, entire GPU
 - $240 * 1.4 \cdot 10^9 = 384 \text{ Giga Load/Store}$



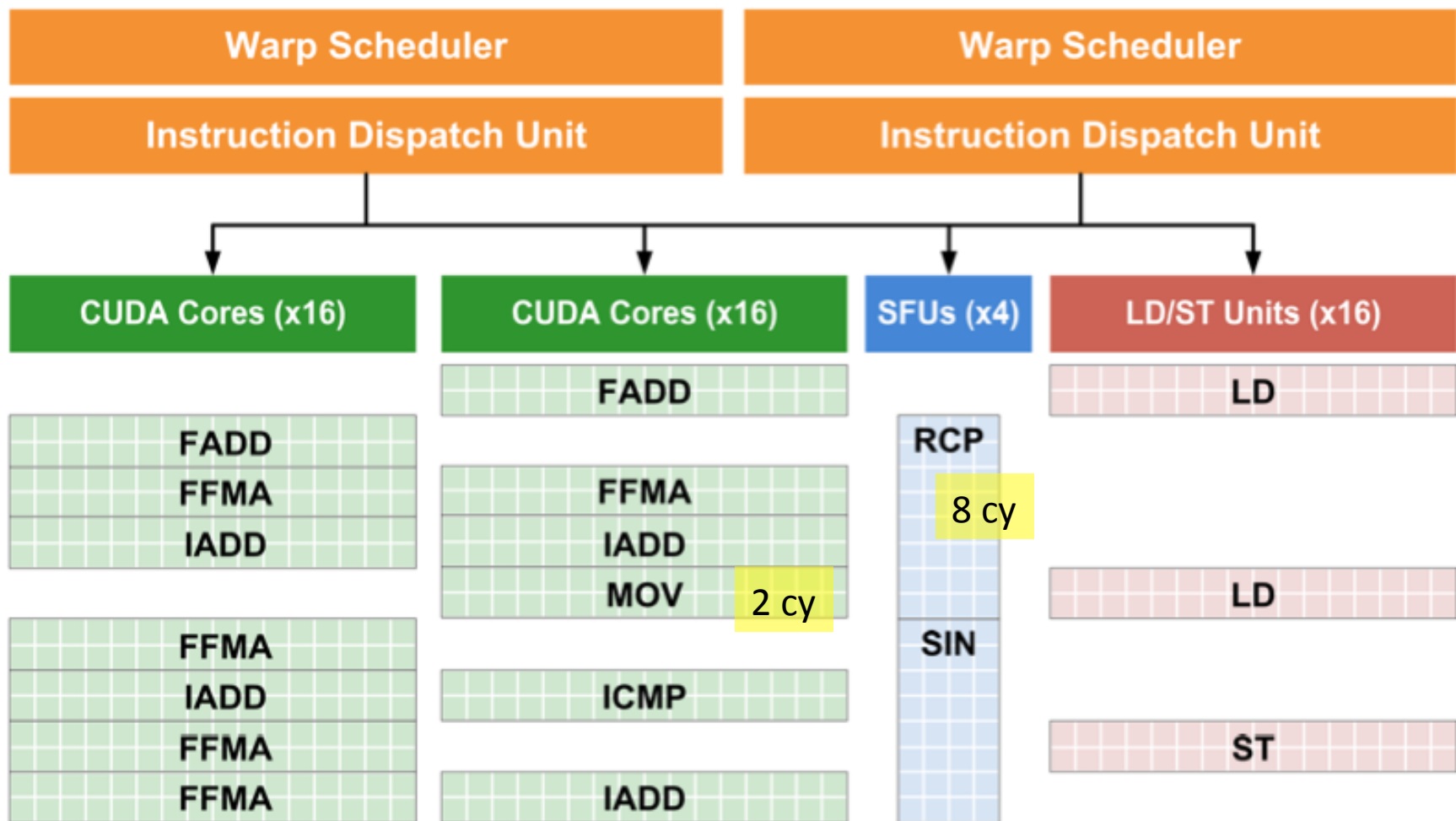


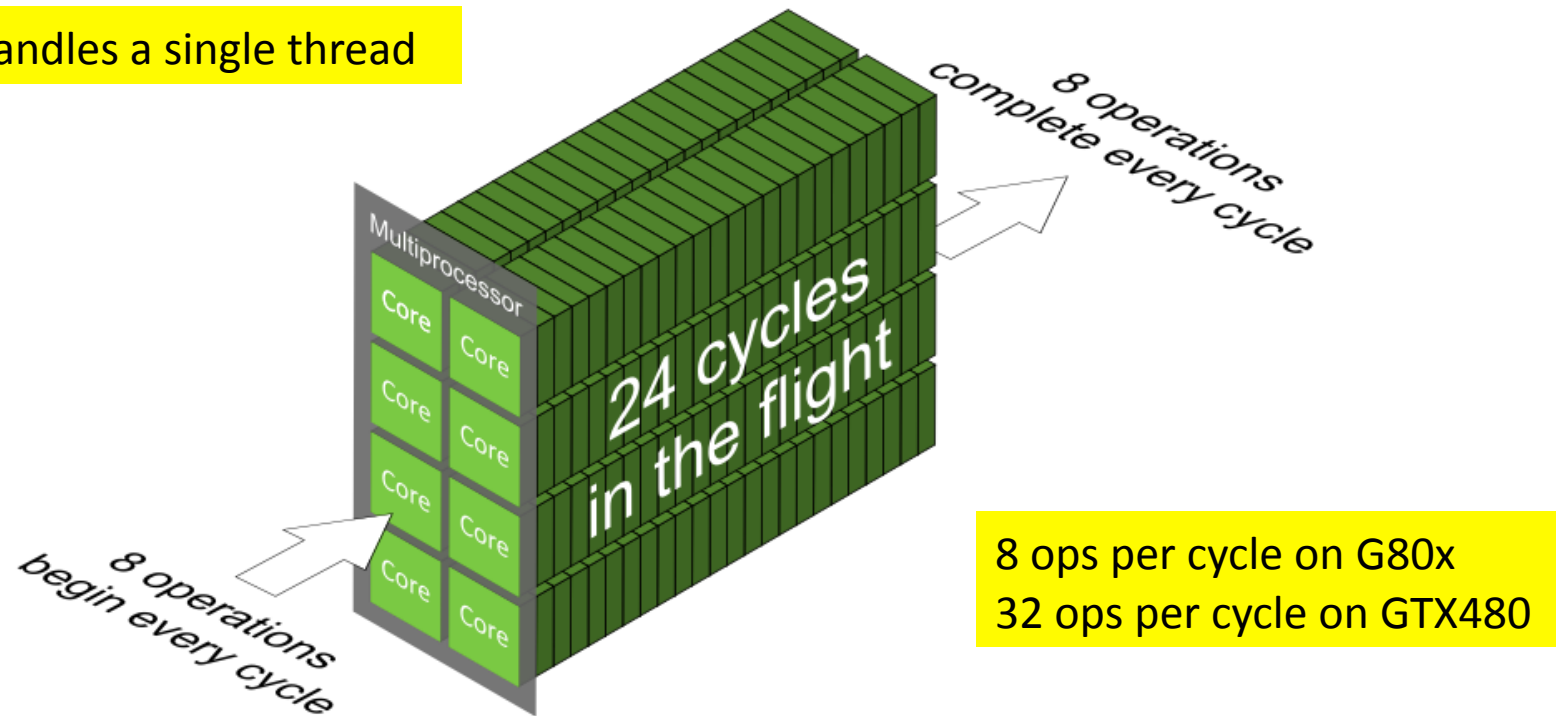
Figure 7. A total of 32 instructions from one or two warps can be dispatched in each cycle to any two of the four execution blocks within a Fermi SM: two blocks of 16 cores each, one block of four Special Function Units, and one block of 16 load/store units. This figure shows how instructions are issued to the execution blocks. (Source: NVIDIA)

Two Fundamental Rules

- A thread *never* executes alone
 - It is the warp that executes, i.e., 32 threads
 - On older GPUs, half-warps, i.e., 16 threads
- Warps execute instructions, then wait
- To add two numbers
 - Two cycles to send these numbers to the FP
 - 24 (18 on Fermi) cycles to wait for the result (= latency)
 - *While the warp is waiting, other warps execute instructions*
 - 24 cycles requires 12 warps to fill up the FP unit (pipelining)

Use Little's law

One core handles a single thread



Needed parallelism = **Latency** x **Throughput**

Pipelining on most machines

Latency

- Computing Latency
 - 2 cycles to issue an add, multiply, or MAD for up to two warps
 - 24 cycles to wait for the result
- Memory Latency
 - 2 cycles to issue a store/load for a warp
 - 400 cycles to retrieve from DRAM (global memory)
- Must do many arithmetic operations to cover I/O latency and slow throughput (compared to FP)
- So need many warps available, waiting for execution

Efficiency

- More art than science
- Need many warps. Options:
 - Single large block, many warps
 - Several blocks, few warps per block
- Barriers: synchronize all threads in a block
 - If no other blocks, time is wasted
 - Often good idea to have between 2 and 4 blocks

Programming Paradigm

- Minimize access to global memory
- Replace multiple access to global memory by
 - Single transfer to shared memory
 - Multiple access to shared memory
- Only 48 Kbytes of shared memory per SM
 - Under control of the user
 - Be frugal

Memory transfers

- Global to shared memory
 - Watch for coalescing problems
 - Each thread of warp have strict address restrictions for maximum throughput (pre-fermi)
 - On Fermi: there is L2/L1 caches, so coalescing much less of a problem
 - Lack of coalescing leads to less efficient transfer (partial serialization)
- On shared memory side: **banking**
 - Different threads of warps must be in different banks for throughput to occur in a single cycle per float (once latency is covered)
 - If this is not the case, transfer occur

Shared memory: banking

- Success 32-bit words are in successive banks
- Each address of a half-warp must be in a separate bank
- If this is not the case, transfer occurs in multiple requests → slowdown

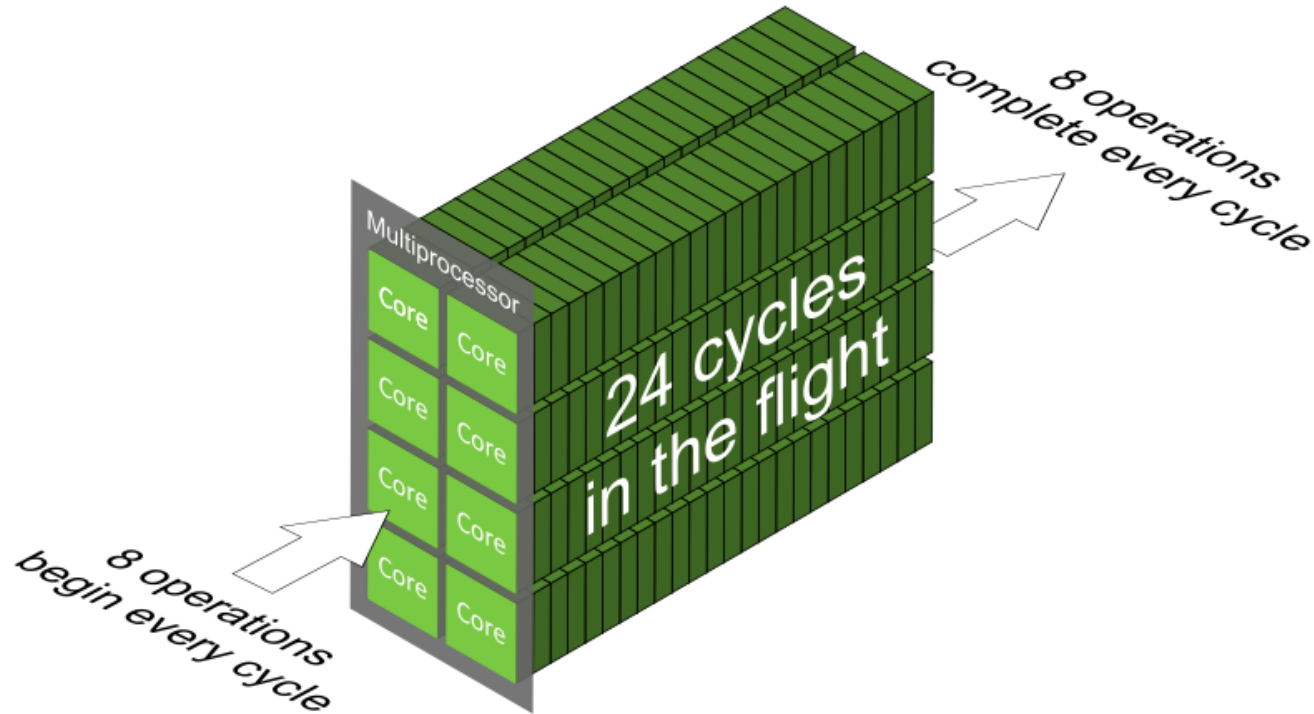
Memory Throughput on Fermi

- Registers: 8 Terabytes/sec
- Shared memory: 1.3 Tbytes/sec
 - 3x slower than registers on pre-Fermi
 - 6x on Fermi (getting worse)
- Global memory: 177 Gbytes/sec = 0.136 Tbytes/sec
 - 58x slower than registers
 - 9.5x slower than shared

Adding two numbers

- Registers -> FP
 - Register throughput is balanced with FP throughput
- Shared memory to FP is too slow
- Global (DRAM/device) memory to FP is way too slow

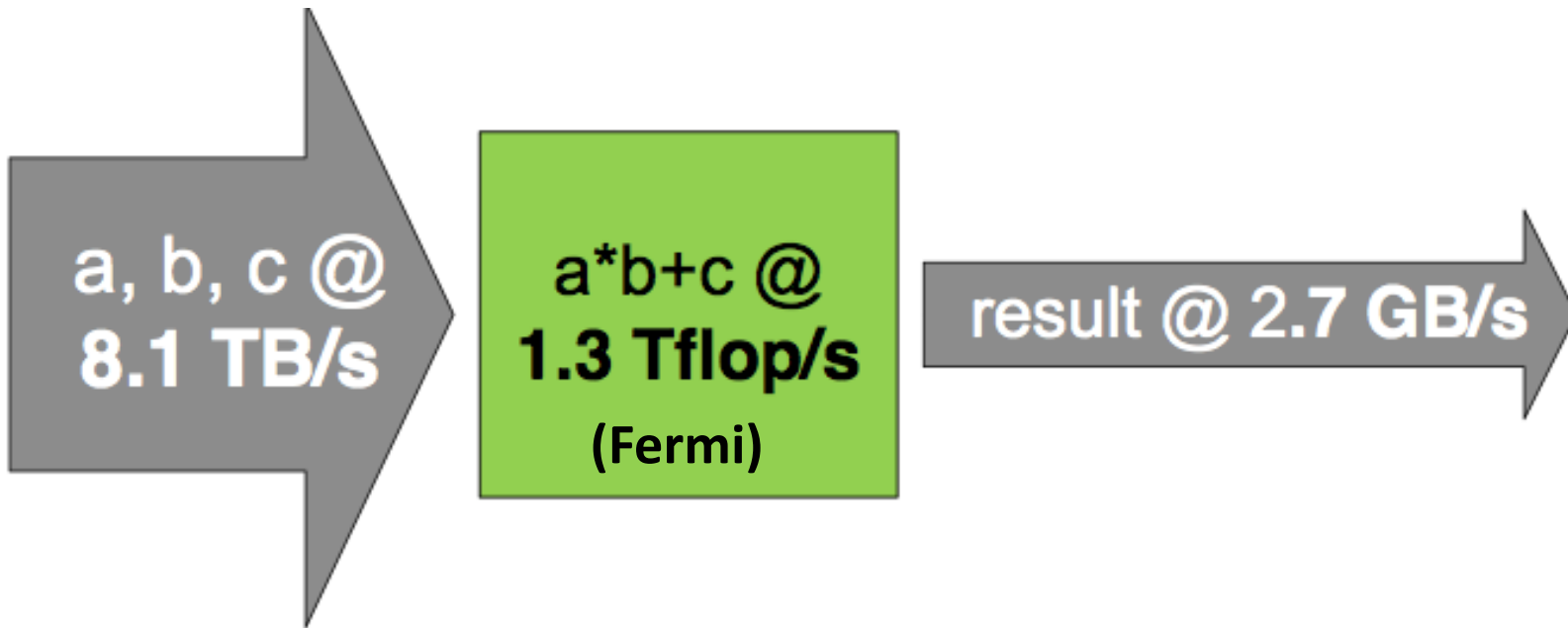
Use Little's law



Needed parallelism = **Latency** x **Throughput**

On Fermi: 32 cores * 18 cycle latency = 576 ops/SM
(required to hide the latency of FP)

Memory/Arithmetic Balancing



Two flops = single MAD (add/multiply)
12 bytes input
4 bytes output
Output bandwidth = 1/3 Input bandwidth

Feed 12 bytes every cycle, per core
 $32 * 12 * 1.4\text{GHz} =$

Simplified Programming

- Jacket (Matlab) (<http://www.accelereyes.com/>)
- Thrust (<http://code.google.com/p/thrust/>)
- Algebra, FFTs, Monte-Carlo CUDA/OpenGL Libraries
- They take care of optimization issues for the high-end user.

Deficiencies of Opencl (2011)

- For the most part, OpenCL maps nicely to CUDA and vice-versa. However,
- No templates, namespaces
 - Difficult to create general frameworks
- Limitations on function arguments
- Constant parameters are hard to manage
- Hard to manage pointers
- No structures of structures on GPU
- Cannot access addresses on GPU
 - This is possible with CUDA
- On average, slower than CUDA
- Single code runs on all platforms
 - Efficiency varies a lot on different GPUs
- More recent, less developed
 - More opportunity

Resources

- GPU Programming Guide 3.1
 - <http://www.nvidia.com/>
- Better Performance at Lower Occupancy
 - By Vasily Volkov, GTC2010
 - www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf
- OpenCL Specification (Khronos)
 - <http://www.khronos.org/registry/cl/>
- Applications ported to CUDA
 - http://www.nvidia.com/object/cuda_home_new.html
- Inside Fermi: Nvidia's HPC Push
 - <http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932&p=3>
- Resource Web page
 - <http://www.sc.fsu.edu/~erlebach/gpus>

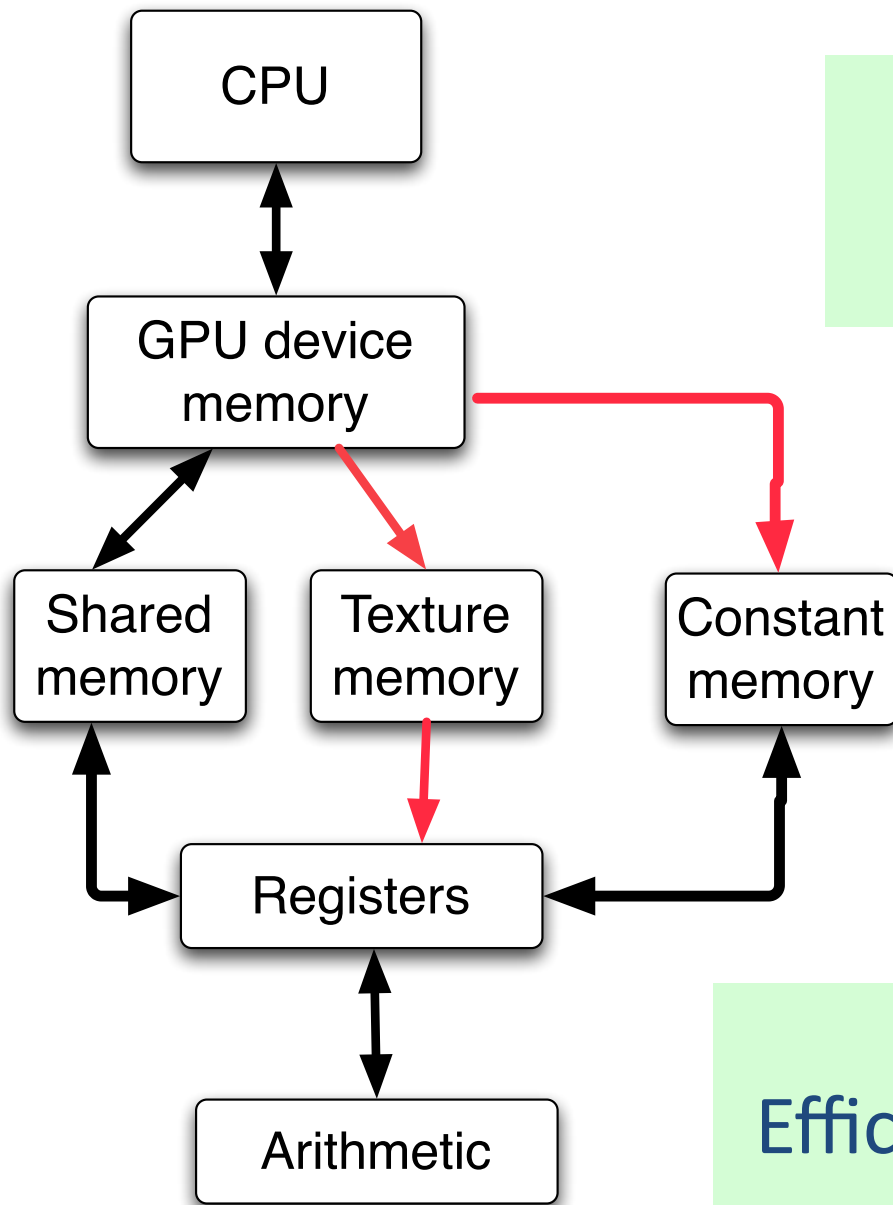
CUDA

Let us return to the Spectral
Finite-Element code

Wednesday, 12:30 pm

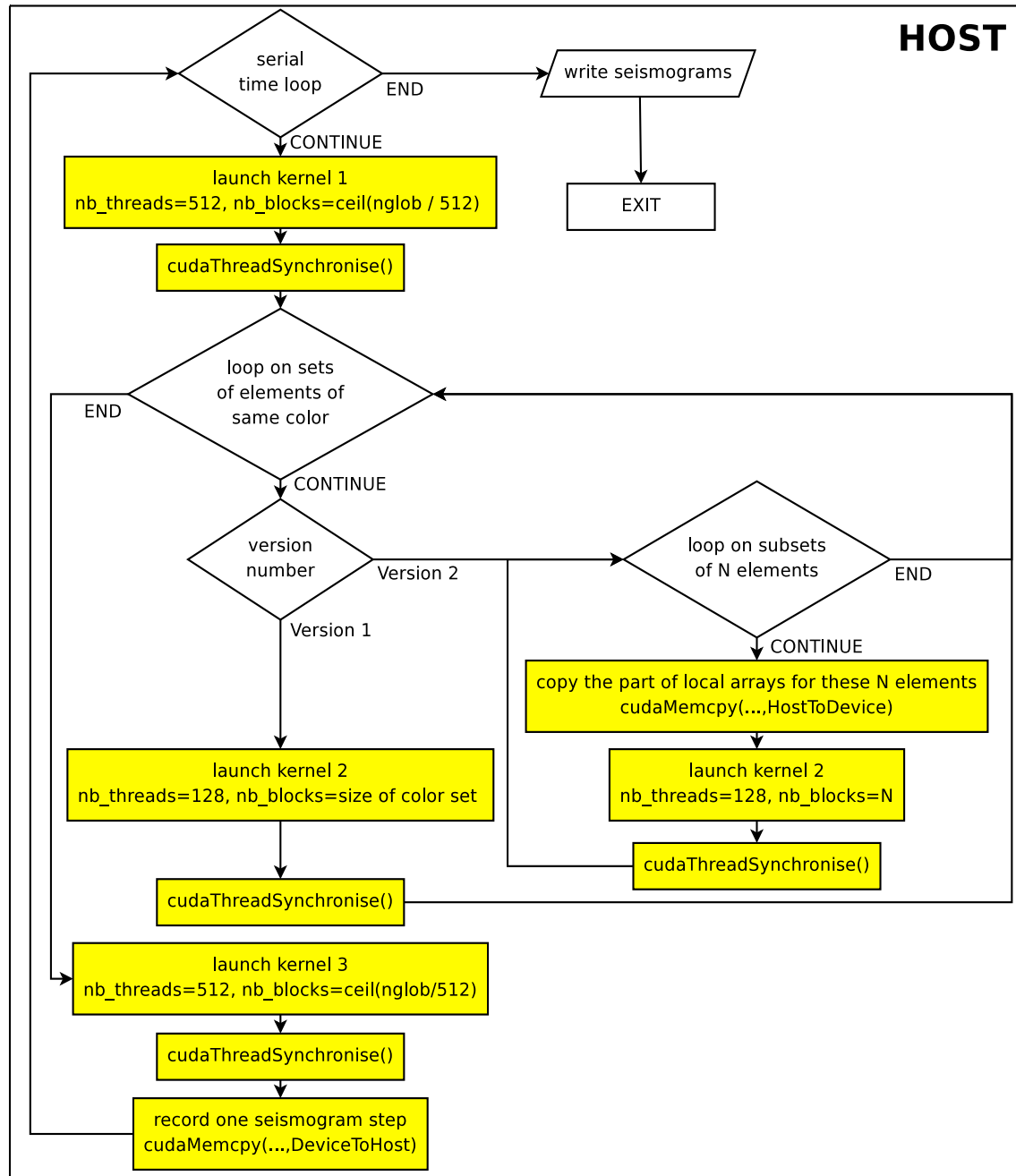
CUDA Implementation

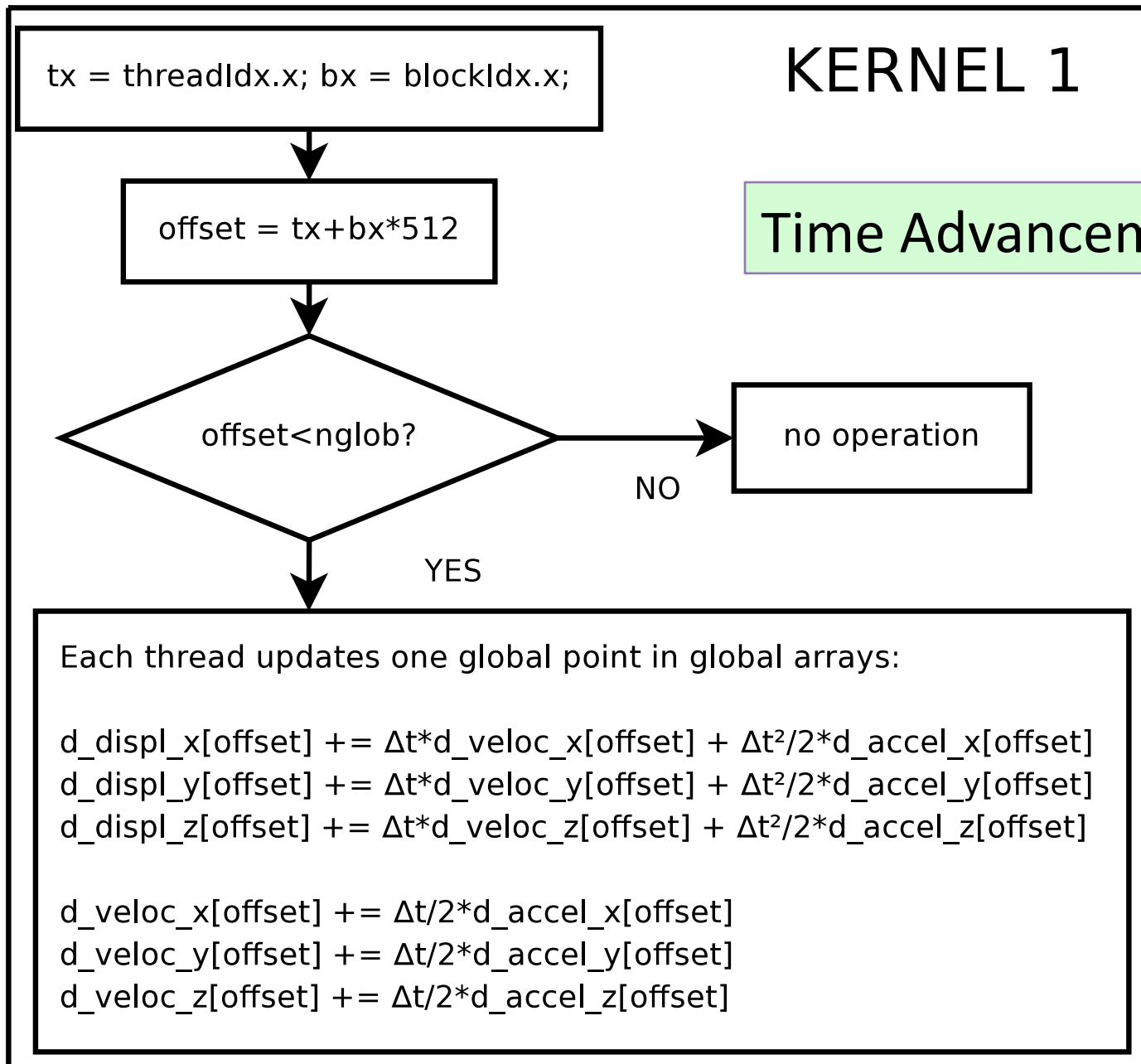
- Each element: $5^3=125$ points
- Each CUDA block = 128 points (waste 3)
- For block max. running concurrently



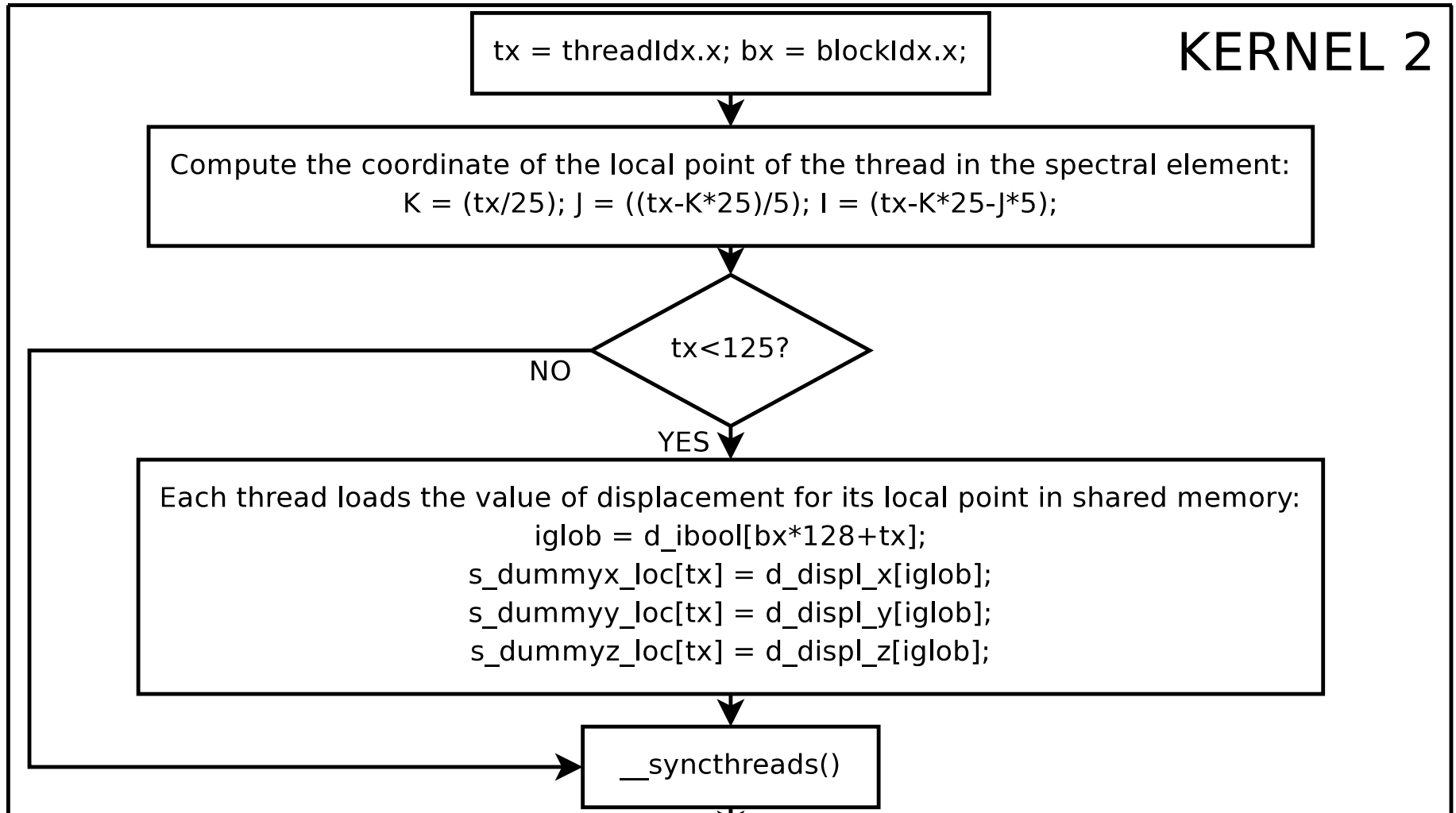
Flow of data on GPU

Efficient Programming

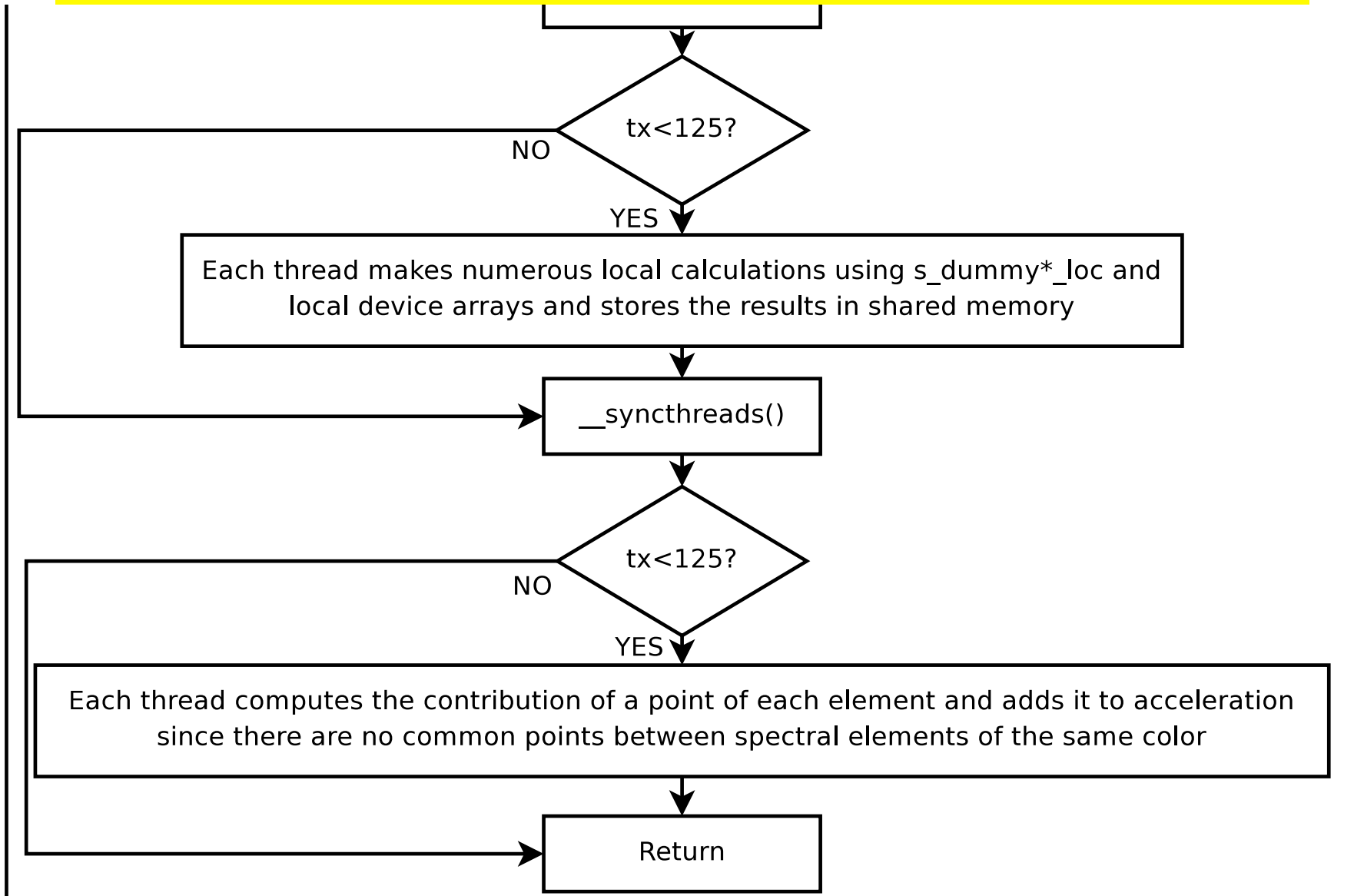




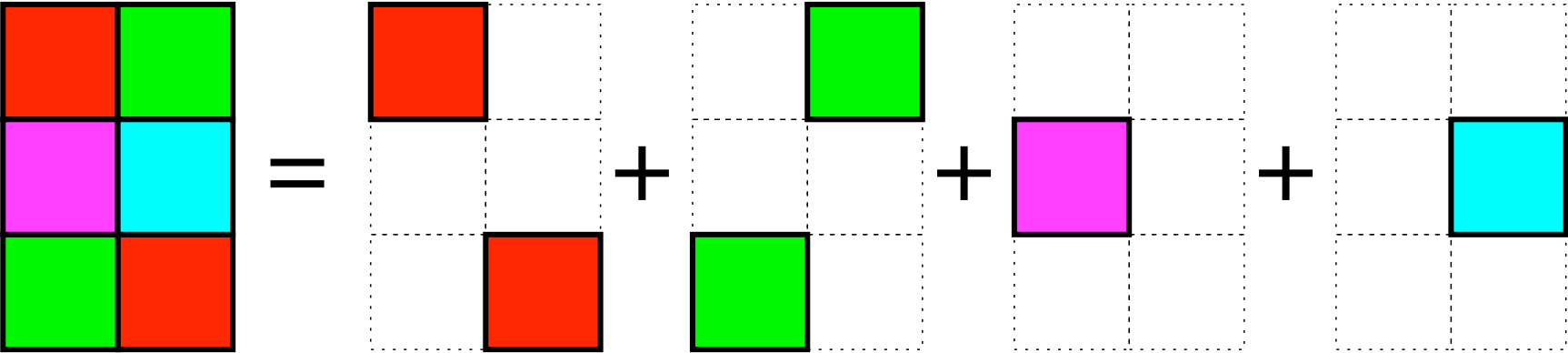
Kernel 2: top half



Kernel 2: bottom half



Coloring



Six
Elements

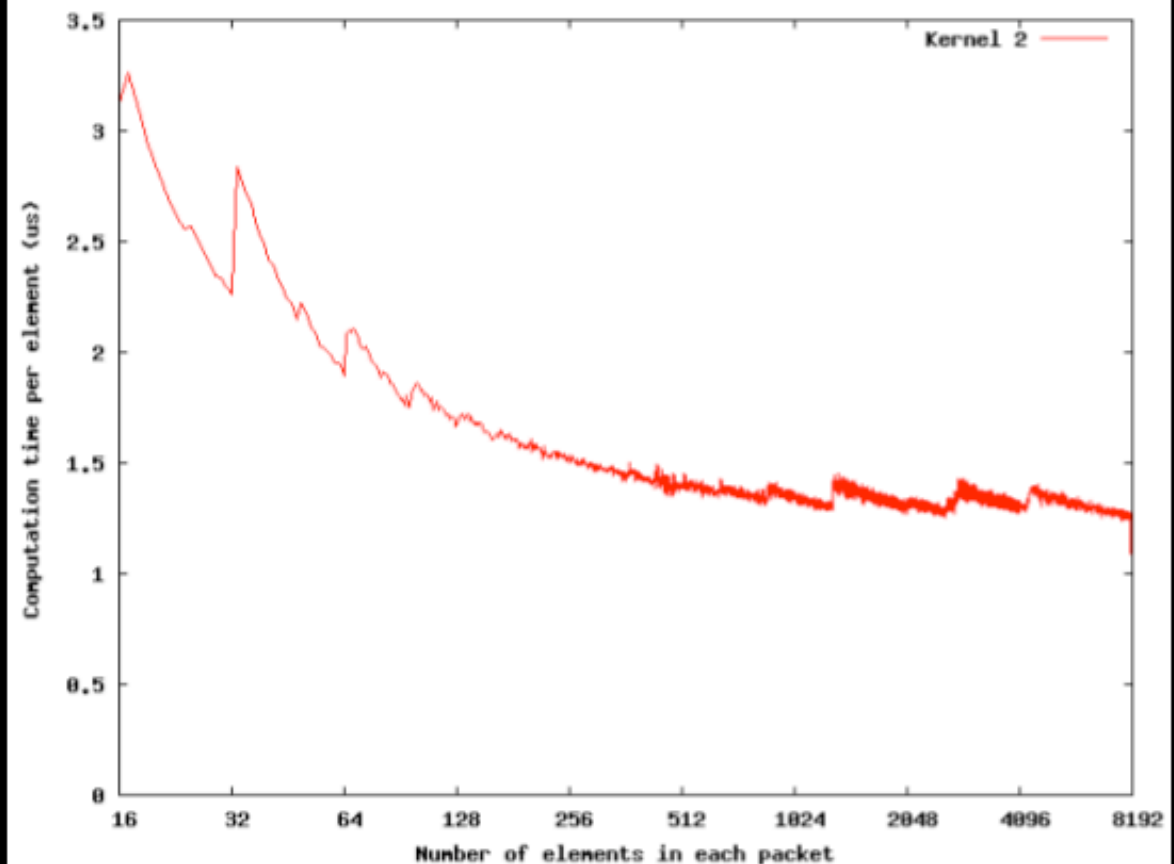
Two elements of given color do
not share the same color

Porting SPECFEM3D on CUDA: results

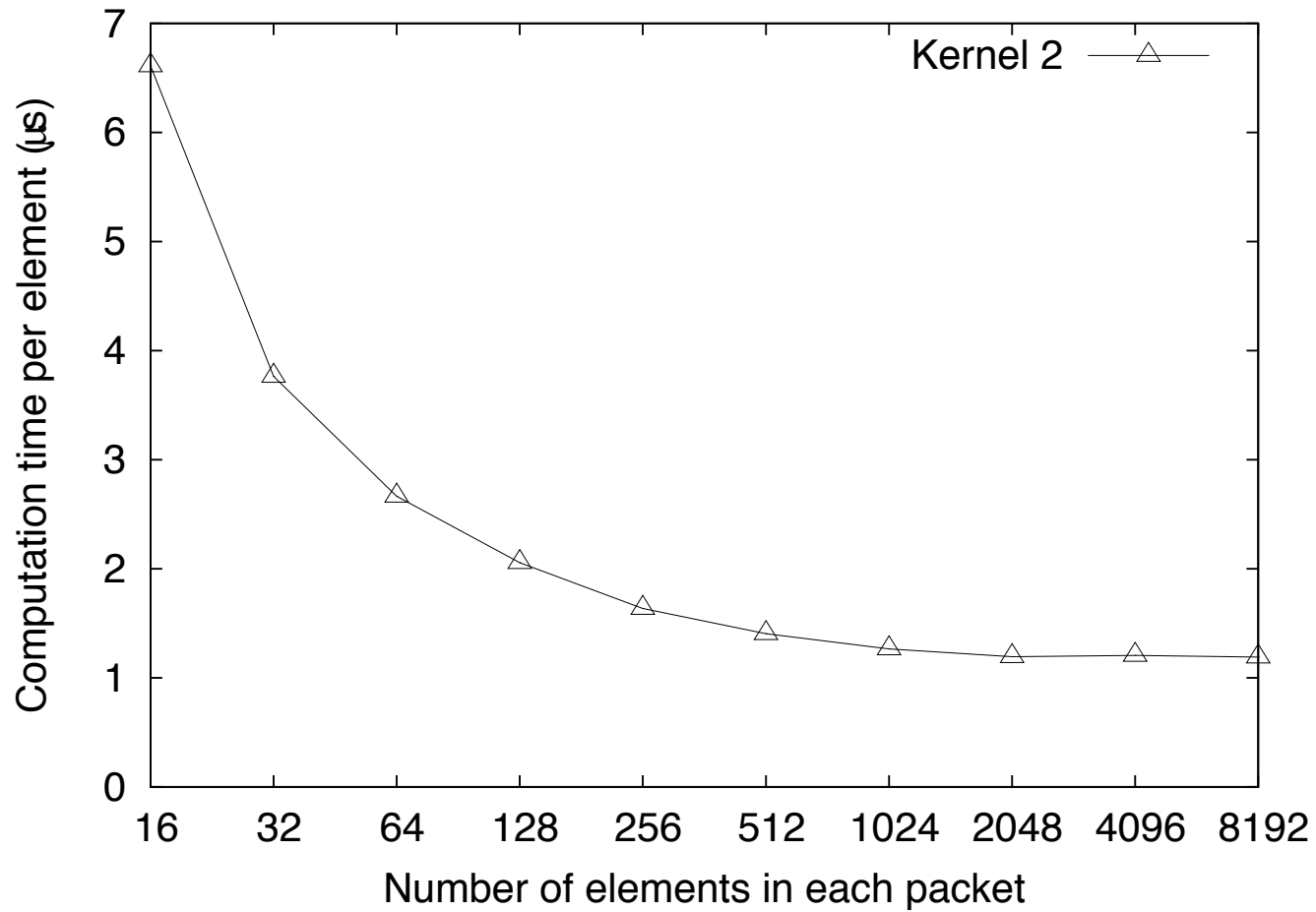
Mesh size	GTX 280		8800 GTX				
	Version 1		Version 1		Version 2		
	Time / element	Speedup	Time / element	Speedup	Time / element	Speedup	Transfert ime
65 MB	0.94 μ s	21.5	1.5 μ s	13.5	4.2 μ s	4.6	68%
405 MB	0.79 μ s	24.8	1.3 μ s	15	3.7 μ s	5.3	68%
633 MB	0.77 μ s	25.3	1.3 μ s	15	3.7 μ s	5.3	67%

● Speedup

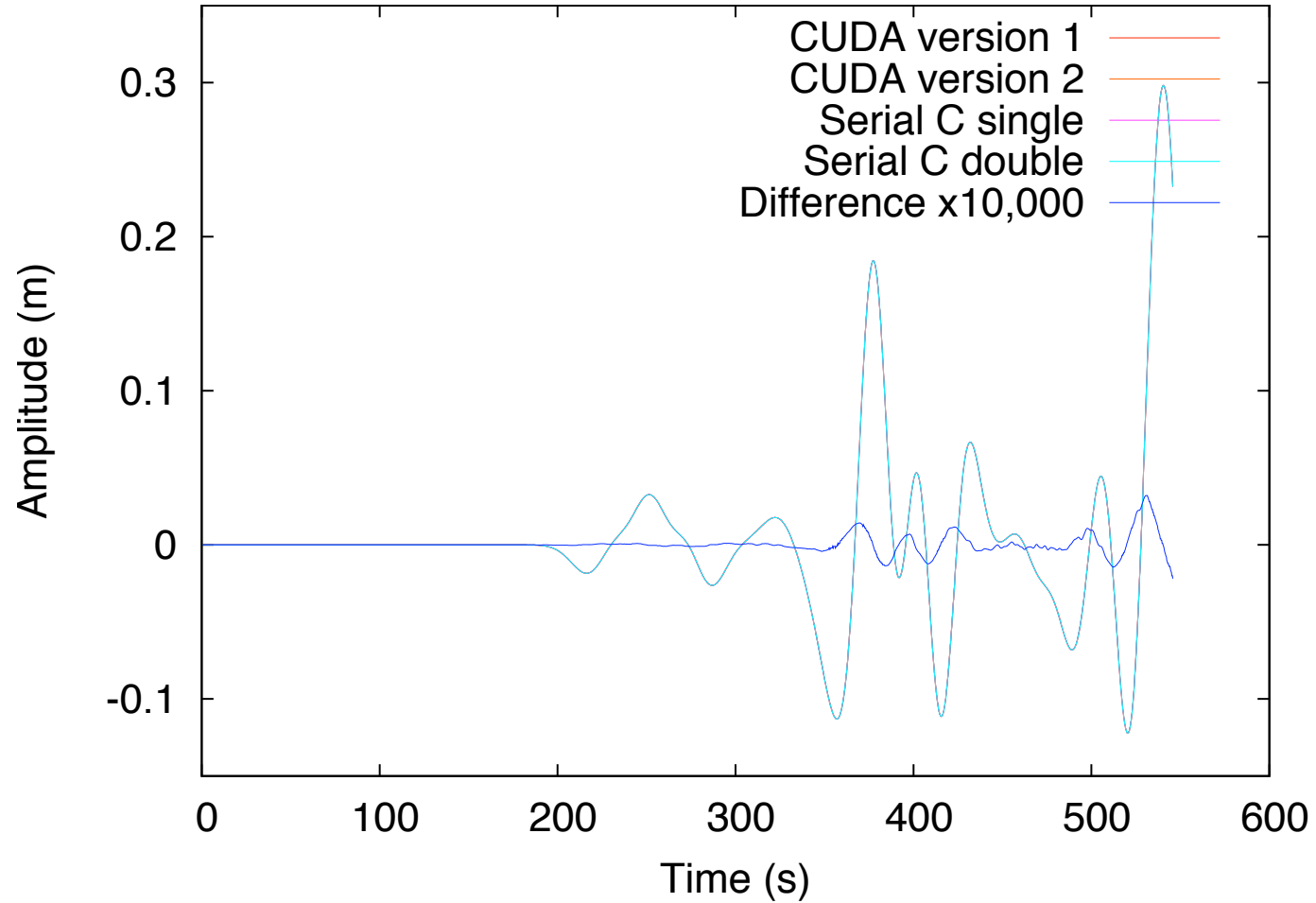
● Performance evolution



Efficiency vs packet size

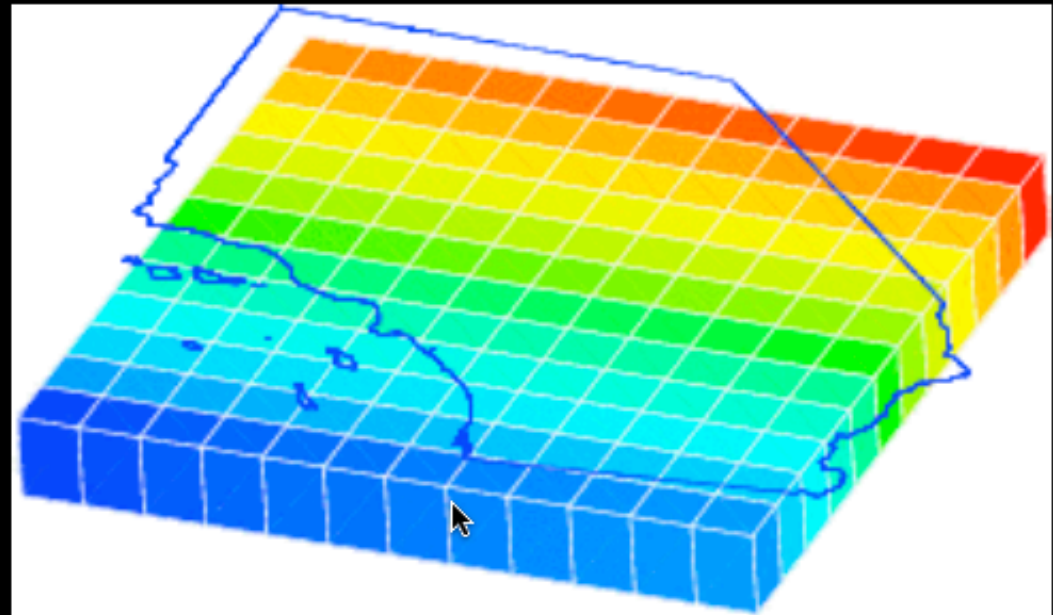
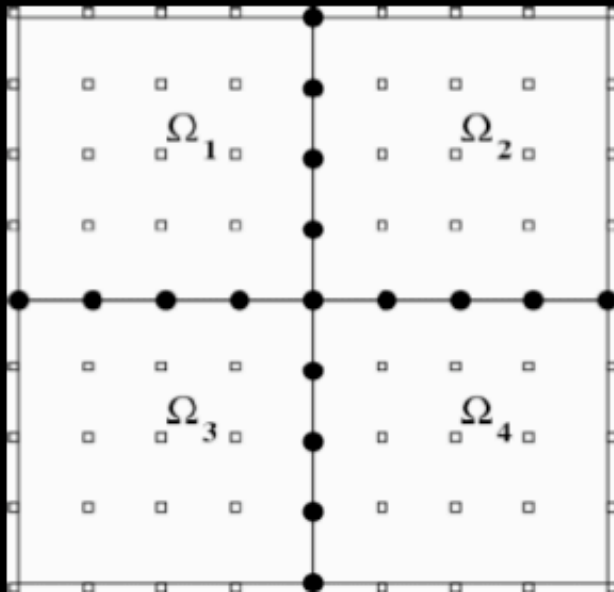
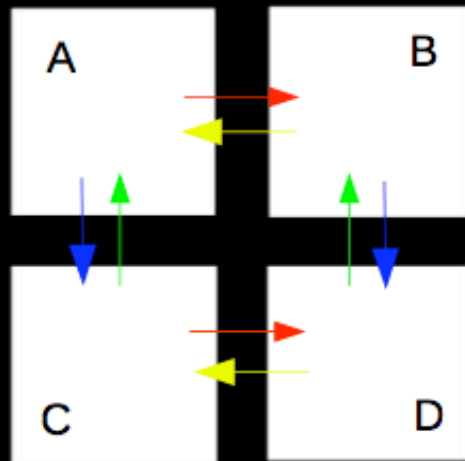


CPU vs GPU versions



Current work: CUDA + MPI

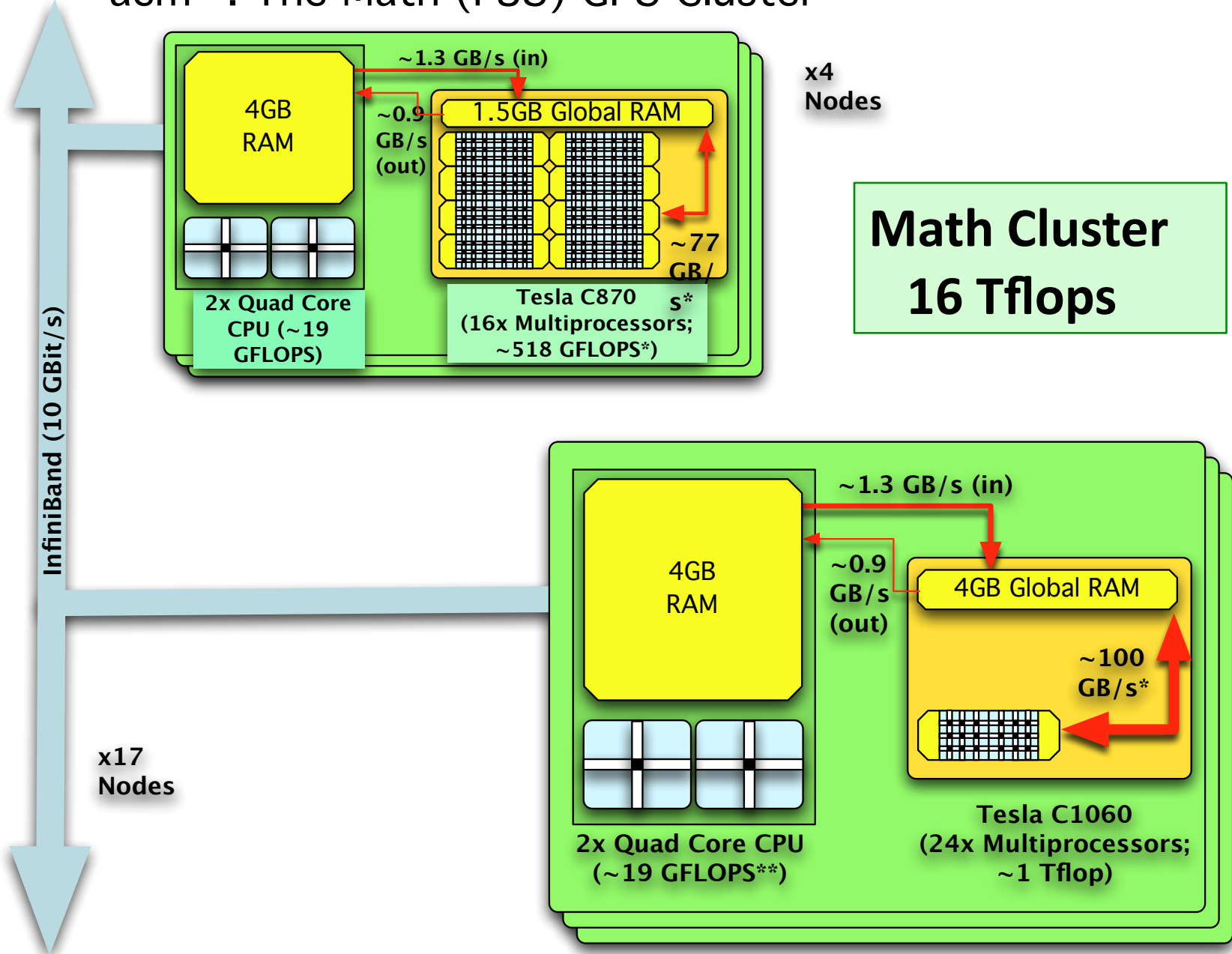
- Previous (classical) communication scheme (blocking MPI)



Communications cost on CPU version ~ 5%,

On the GPU version, with a speedup of 25, communication cost ~58%

"acm" : The Math (FSU) GPU Cluster



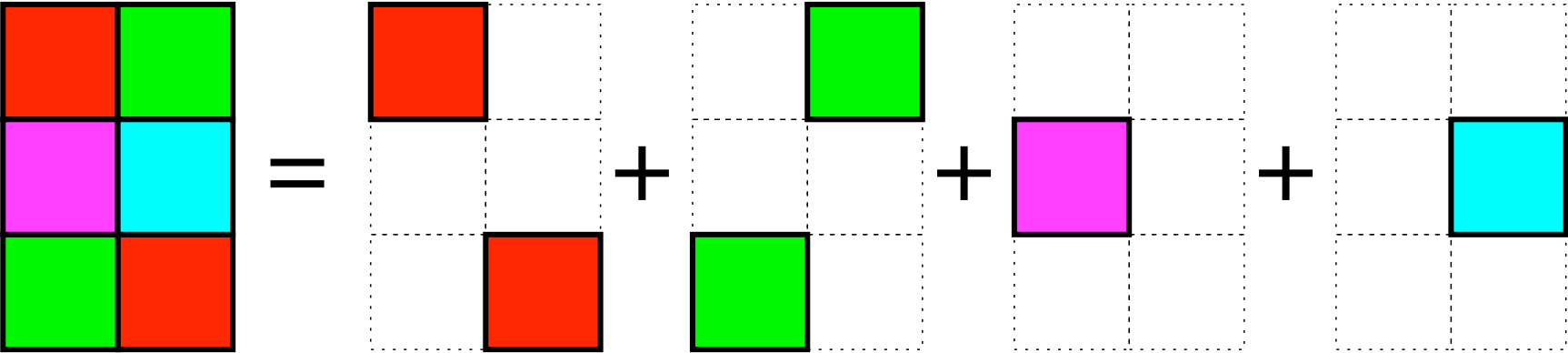
Conclusion

- GPUs have higher rate of performance increase over time than CPUs
 - always appealing as “research for the future”
- In certain applications GPUs are 15 to 60 times faster than CPUs for **low precision**
- For certain floating point applications GPU’s and CPU’s performance is comparable
 - can be used as coprocessor
- GPUs are often constrained in memory, but
- It is feasible to use GPUs for numerical simulations
 - Languages: CUDA, RapidMind (cell processors and GPUs)
- Calculations in double precisions should be avoided
 - use mixed precision calculations

Resources

- SC Visualization Lab
 - SC 4 HPs with GeForce 7900, 8800, 280GTX cards
 - 1 HP 9100 with Quadro 5600 for GPU computation and Stereo viewing
- Math resource:
 - 20 processors with 20 Tesla cards.

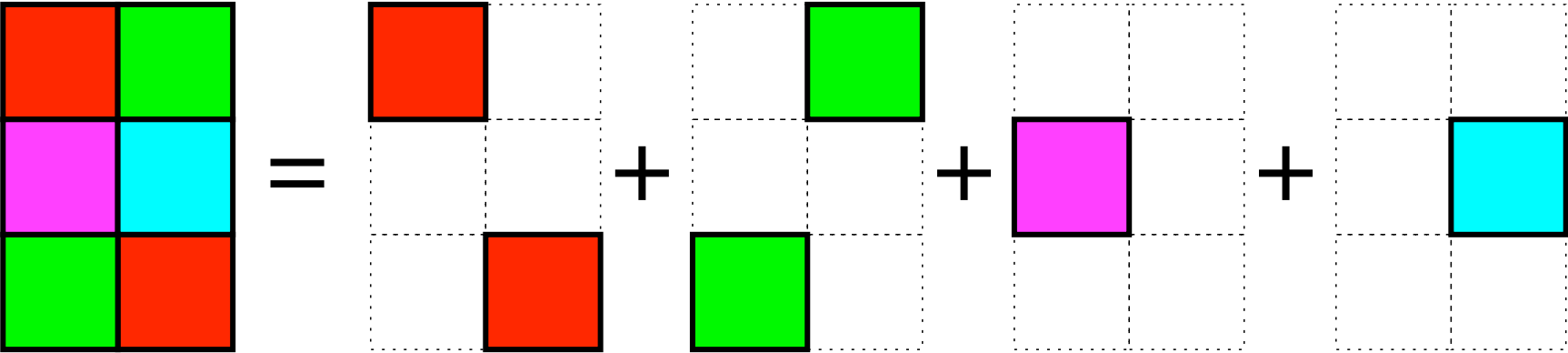
Coloring



Six
Elements

Two elements of given color do
not share the same color

Coloring



Six
Elements

Two elements of given color do
not share the same color