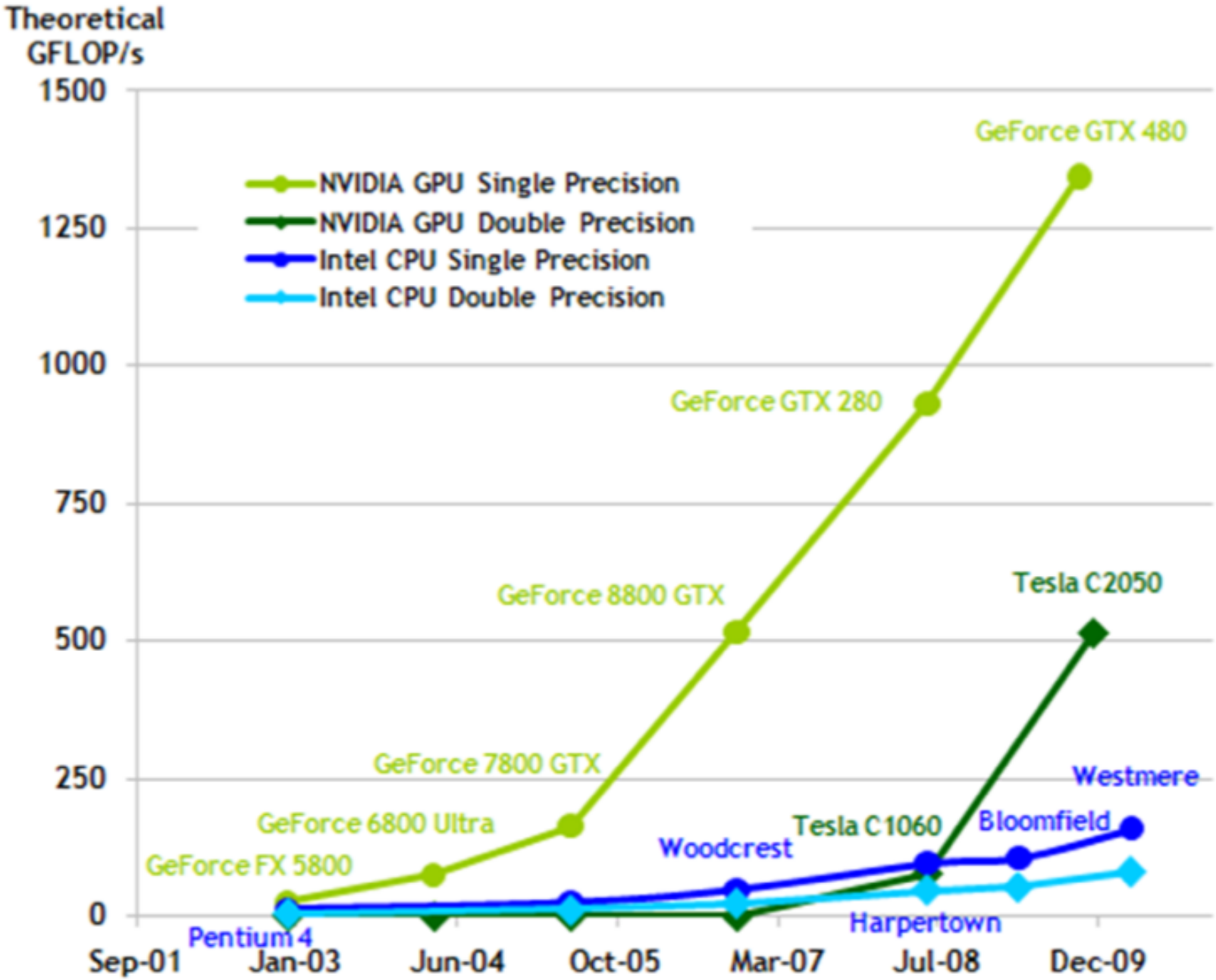
Programming GPUs (CUDA and OpenCL) Part 1: architecture

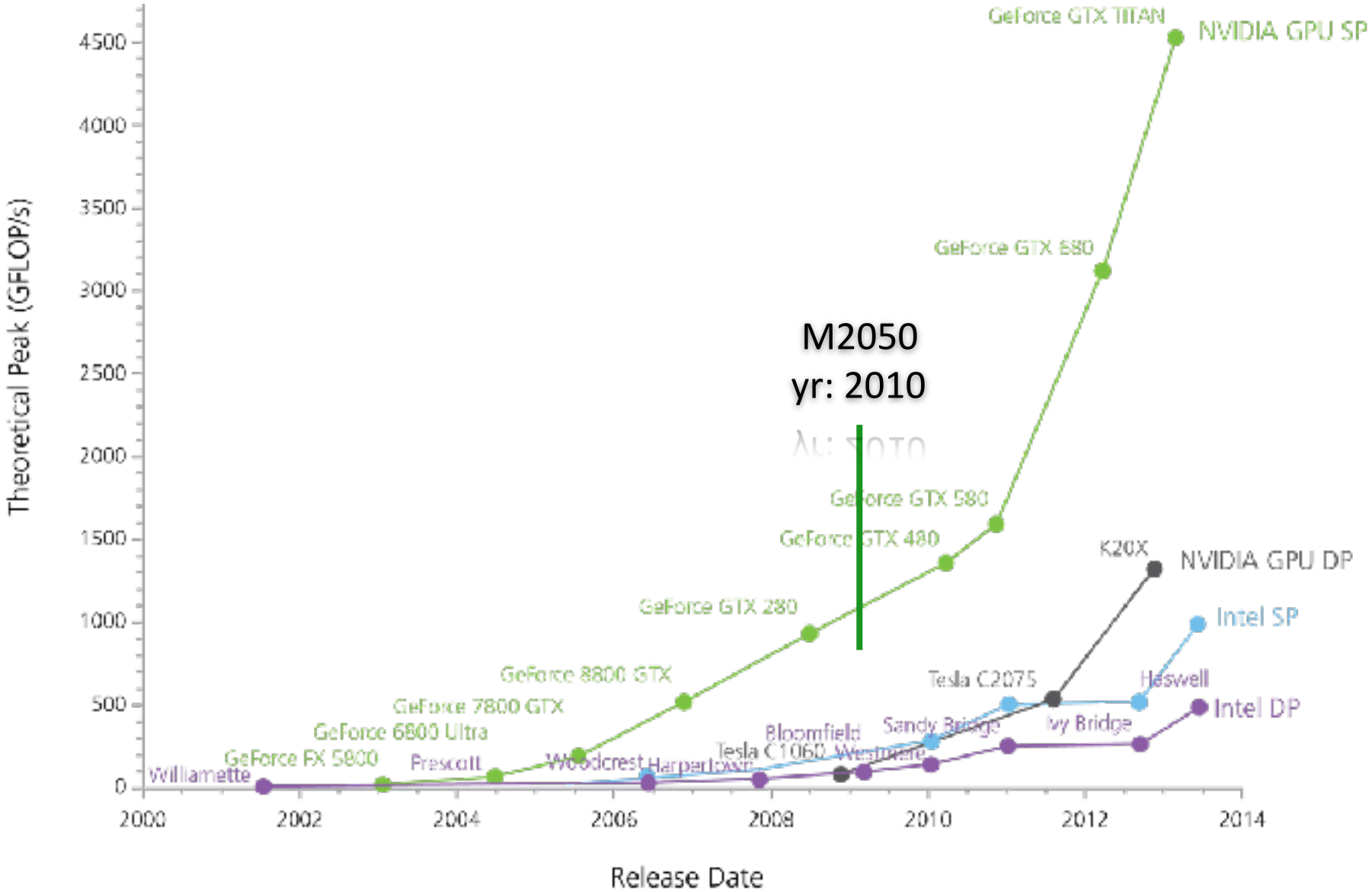
Gordon Erlebacher

Nov. 8, 2010 (original talk)

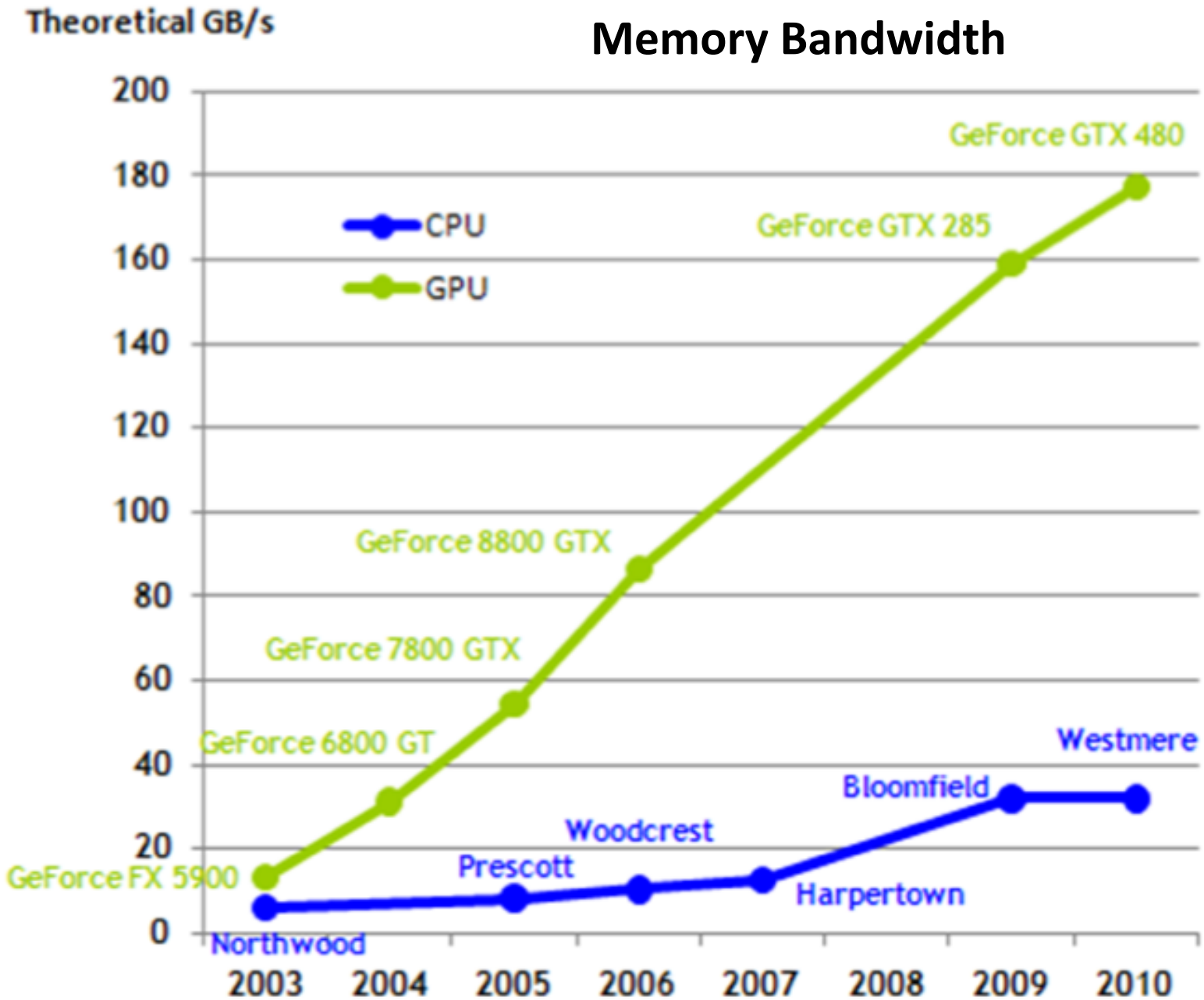
April 20, 2017 (current talk)


Theoretical GFLOP/s
1500
1250
1000
750
500
250
0
NVIDIA GPU Single Precision
NVIDIA GPU Double Precision
Intel CPU Single Precision
Intel CPU Double Precision
GeForce GTX 480
GeForce GTX 280
GeForce 8800 GTX
Tesla C2050
GeForce 7800 GTX
Westmere
GeForce 6800 Ultra
Tesla C1060
Bloomfield
Woodcrest
GeForce FX 5800
Harpertown
Pentium 4
Sep-01
Jan-03
Jun-04
Oct-05
Mar-07
Jul-08
Dec-09

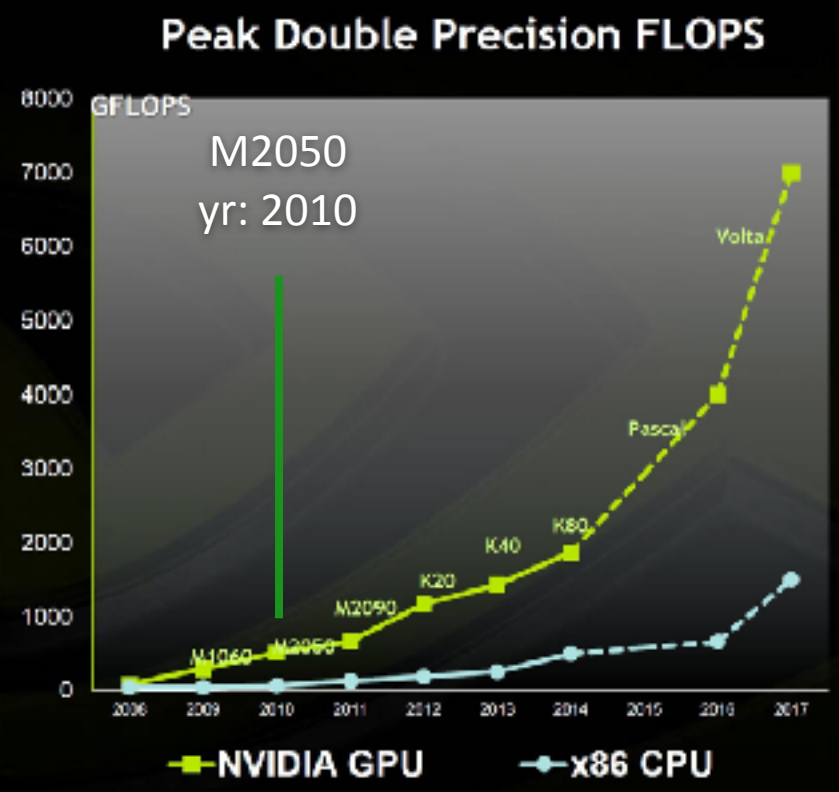
Peak flops (2000-2014)

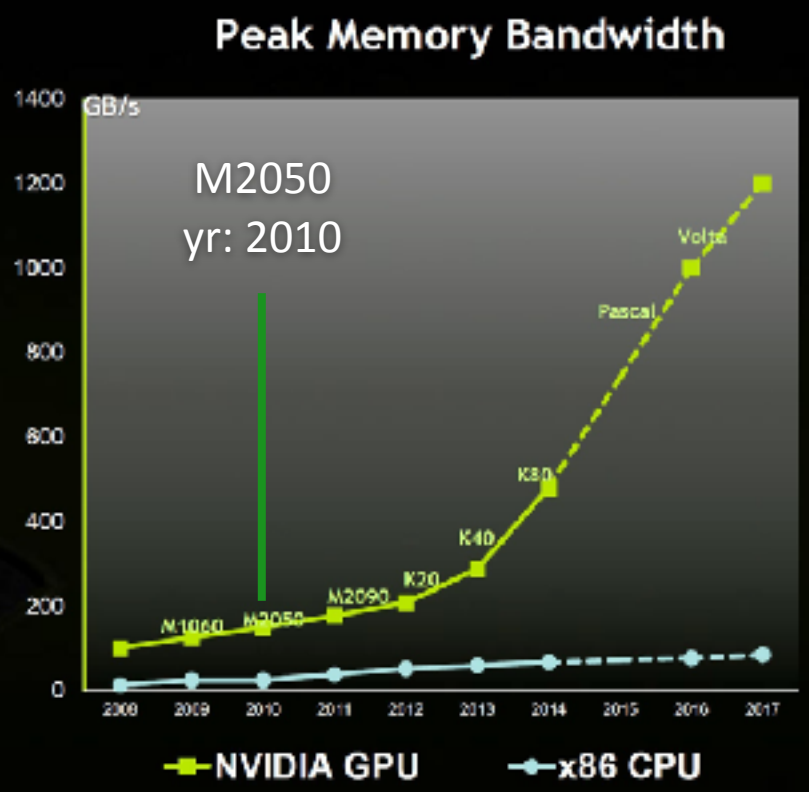


Memory Bandwidth



GPU Motivation (I): Performance Trends

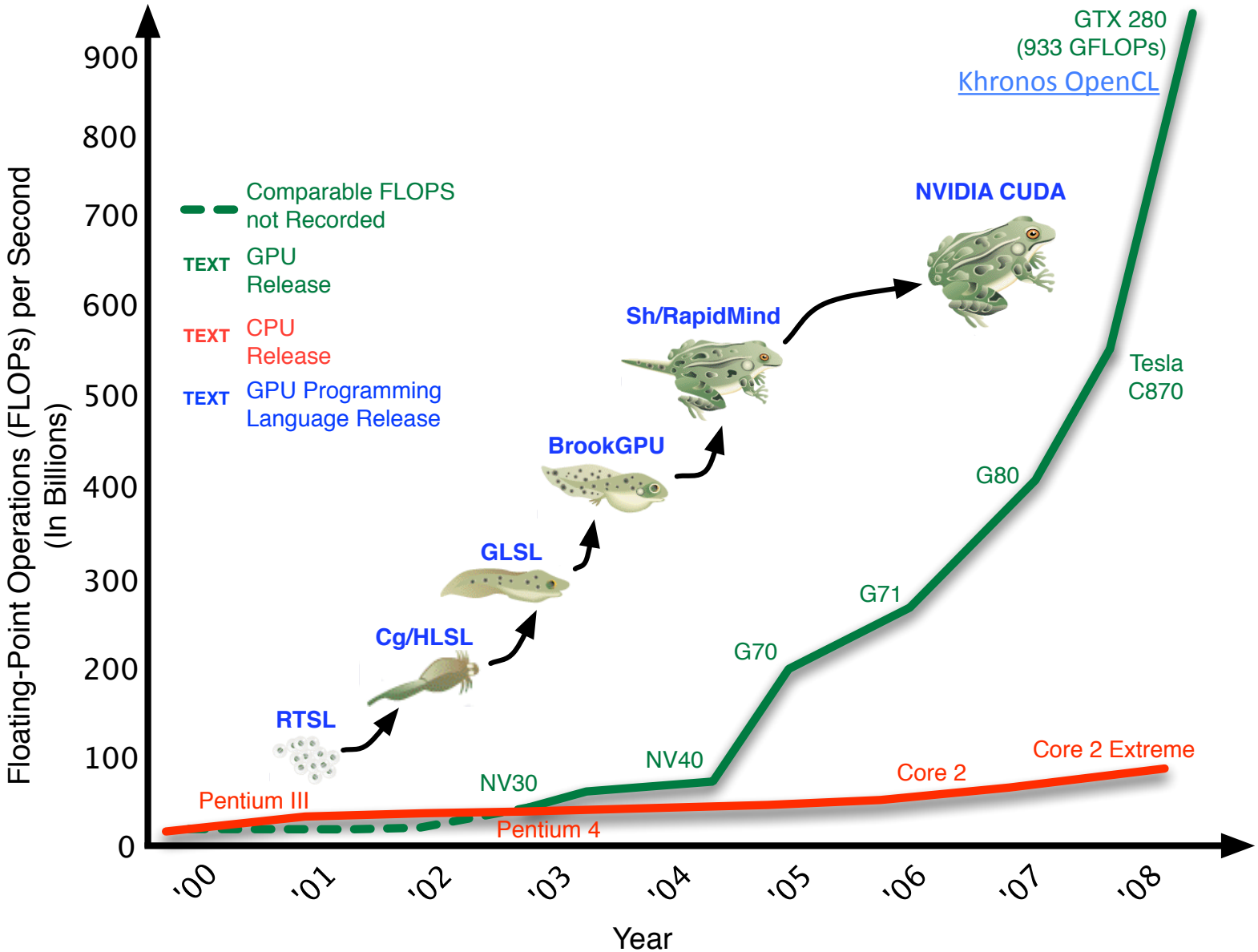




7

Evolution GPU Languages

Fermi, GTX480



GPU Architectures:

	Tesla	Fermi	Kepler	Maxwell	Pascal	
Card	GeForce 100-300	GeForce 400-500	GeForce 600-700	GeForce 900	GeForce 10	
Year	2007-2010	2010-2011	2012-2014	2014-2016	1700-10000	
Memory Bandwidth	10-200	10-300	20-600	14-300	100-500	GB/s
Single Precision	20-500	100-3000	100-8000	1000-6000	1700-10000	GFlops
Double Precision	N/A	N/A	14-2000	32-200	54-300	GFlops
Cost	N/A	$60-$500	$45-$3000	$150-$1000	$100-$700	

SOURCE: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#GeForce_600_series

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

GTX 480 (graphics Fermi) (used in my 2010 talk

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:

Table 2. Compute Capabilities: GK110 vs GM200 vs GP100

GPU	Kepler GK110	Maxwell GM200	Pascal GP100
Compute Capability	3.5	5.2	6.0
Threads / Warp	32	32	32
Max Warps / Multiprocessor	64	64	64
Max Threads / Multiprocessor	2048	2048	2048
Max Thread Blocks / Multiprocessor	16	32	32
Max 32-bit Registers / SM	65536	65536	65536
Max Registers / Block	65536	32768	65536
Max Registers / Thread	255	255	255
Max Thread Block Size	1024	1024	1024
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB

Table 1. Tesla P100 Compared to Prior Generation Tesla products

Tesla Products	Tesla K40	Tesla M40	Tesla P100
GPU	GK110 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)
SMs	15	24	56
TPCs	15	24	28
FP32 CUDA Cores / SM	192	128	64
FP32 CUDA Cores / GPU	2880	3072	3584
FP64 CUDA Cores / SM	64	4	32
FP64 CUDA Cores / GPU	960	96	1792
Base Clock	745 MHz	948 MHz	1328 MHz
GPU Boost Clock	810/875 MHz	1114 MHz	1480 MHz
Peak FP32 GFLOPs[1]	5040	6840	10600
Peak FP64 GFLOPs[1]	1680	210	5300
Texture Units	240	192	224
Memory Interface	384-bit GDDR5	384-bit GDDR5	4096-bit HBM2
Memory Size	Up to 12 GB	Up to 24 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB
Register File Size / SM	256 KB	256 KB	256 KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB
TDP	235 Watts	250 Watts	300 Watts
Transistors	7.1 billion	8 billion	15.3 billion
GPU Die Size	551 mm^2	601 mm^2	610 mm^2
Manufacturing Process	28-nm	28-nm	16-nm FinFET

Cost $1200

Tesla P100

Pascal architecture

SPECIFICATIONS

GPU Architecture	**NVIDIA Pascal**
NVIDIA CUDA® Cores	**3584**
Double-Precision Performance	**5.3 TeraFLOPS**
Single-Precision Performance	**10.6 TeraFLOPS**
Half-Precision Performance	**21.2 TeraFLOPS**
GPU Memory	**16 GB CoWoS HBM2**
Memory Bandwidth	**732 GB/s**
Interconnect	**NVIDIA NVLink**
Max Power Consumption	**300 W**
ECC	**Native support with no capacity or performance overhead**
Thermal Solution	**Passive**
Form Factor	**SXM2**
Compute APIs	**NVIDIA CUDA, DirectCompute, OpenCL™, OpenACC**

TeraFLOPS measurements with NVIDIA GPU Boost™ technology

Tesla 1060: Nov. 2008

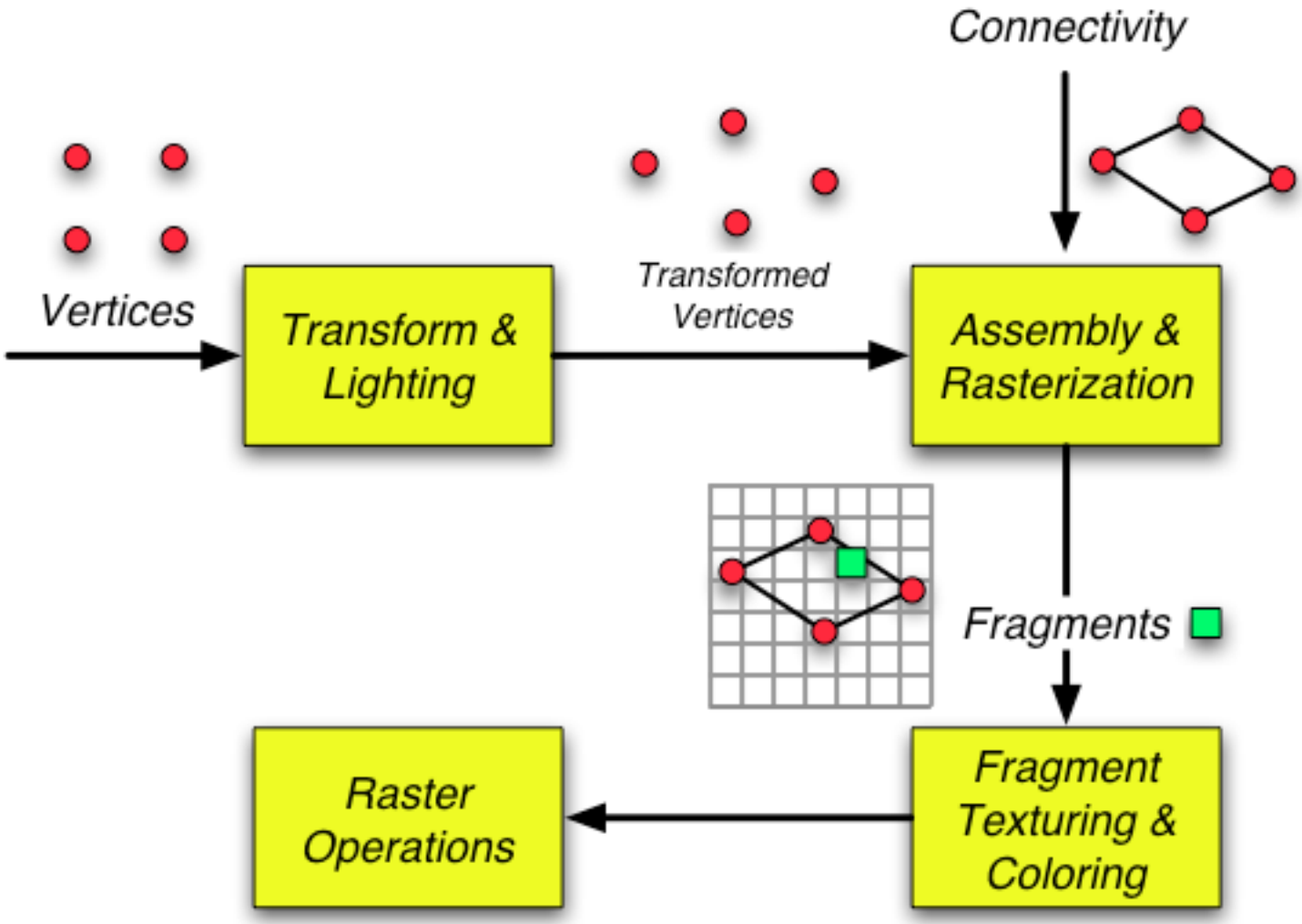
Form Factor	10.5" x 4.376", Dual Slot
# of Tesla GPUs	1
# of Streaming Processor Cores	240
Frequency of processor cores	1.3GHz
Single Precision floating point performance (peak)	933
Double Precision floating point performance (peak)	78
Floating Point Precision	IEEE 754 single & double
Total Dedicated 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

Graphic Card Evolution on PC

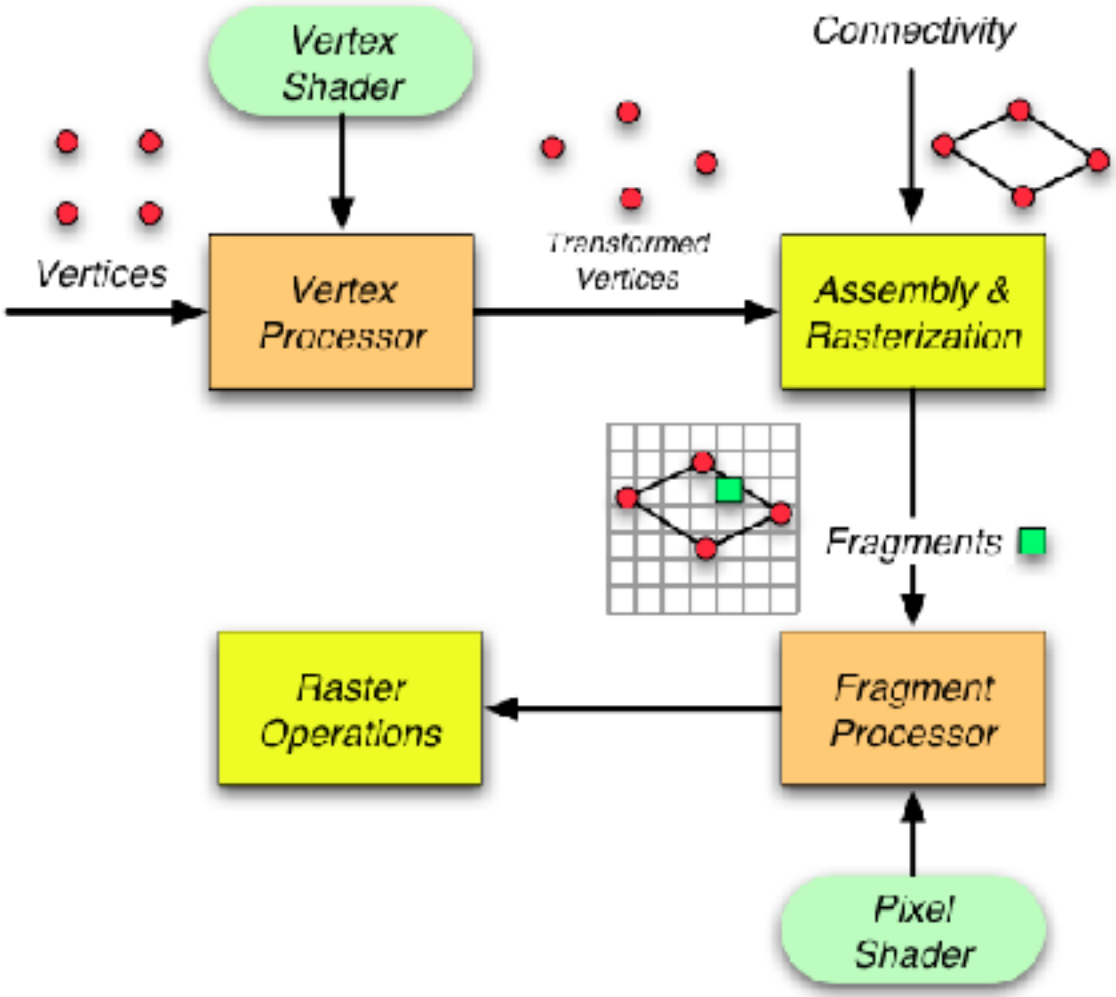
- **Increased graphics functionality is relegated to the graphics card**
  - < 1995 : fixed pipeline. Specialized workstations: Silicon Graphics ($100k) Amiga: first PC with graphics controlled by special card
  - 1995-1998: 2D acceleration, textures and z-buffer
  - 1998: multi-textures
  - 1999-2000: 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)

http://en.wikipedia.org/wiki/Comparison_of_NVIDIA_Graphics_Processing_Units

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];
   }
}
```

odata(i,j) = idata(j,i)

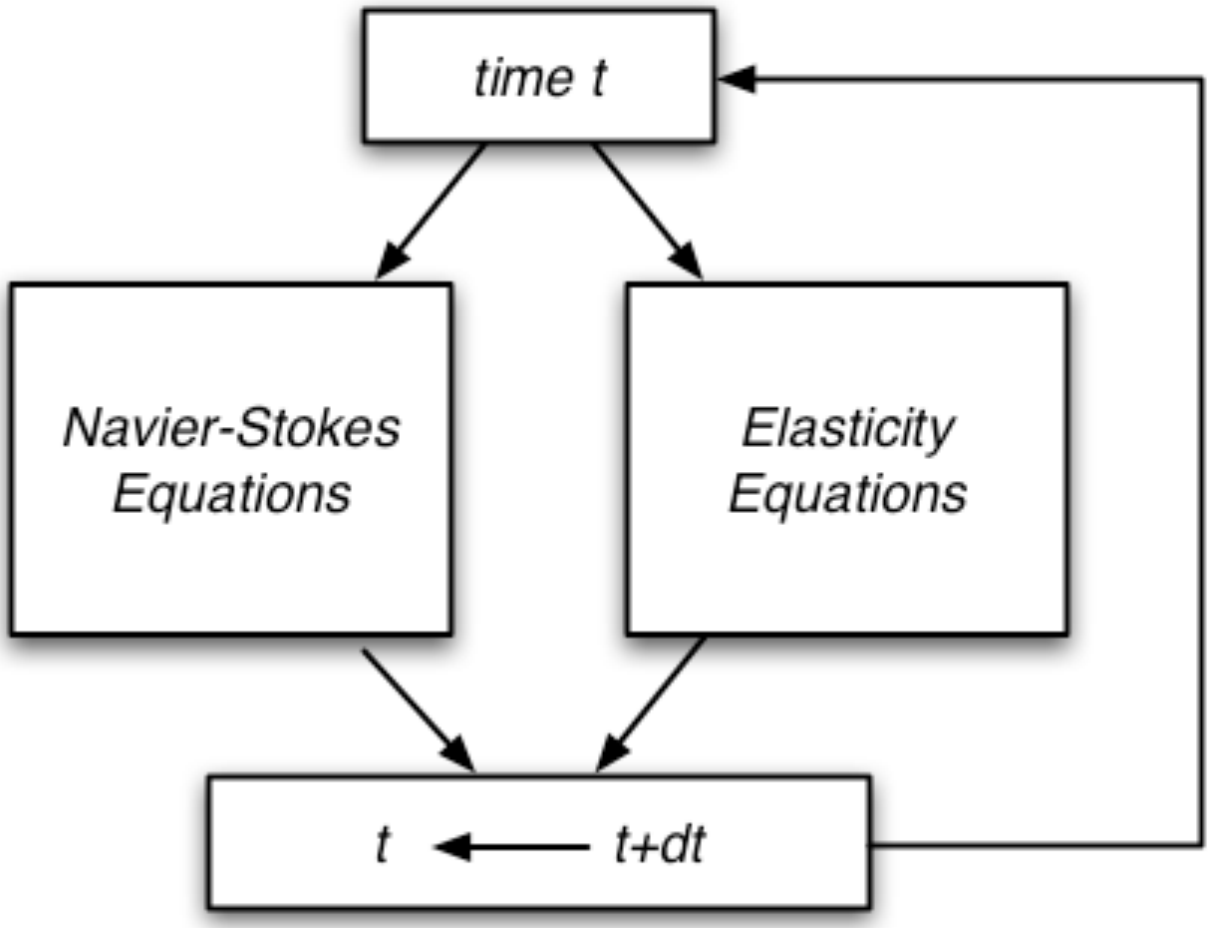
Transpose on today's GPUs

- I have not done the experiment, but I would expect simpler code with higher efficiency
- **Higher efficiency:**
  - higher fraction of actual to peak performance

Parallelism

Task parallelism
Data parallelism
SIMD
MIMD
Stream

Task Parallelism



Data Parallelism

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

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

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

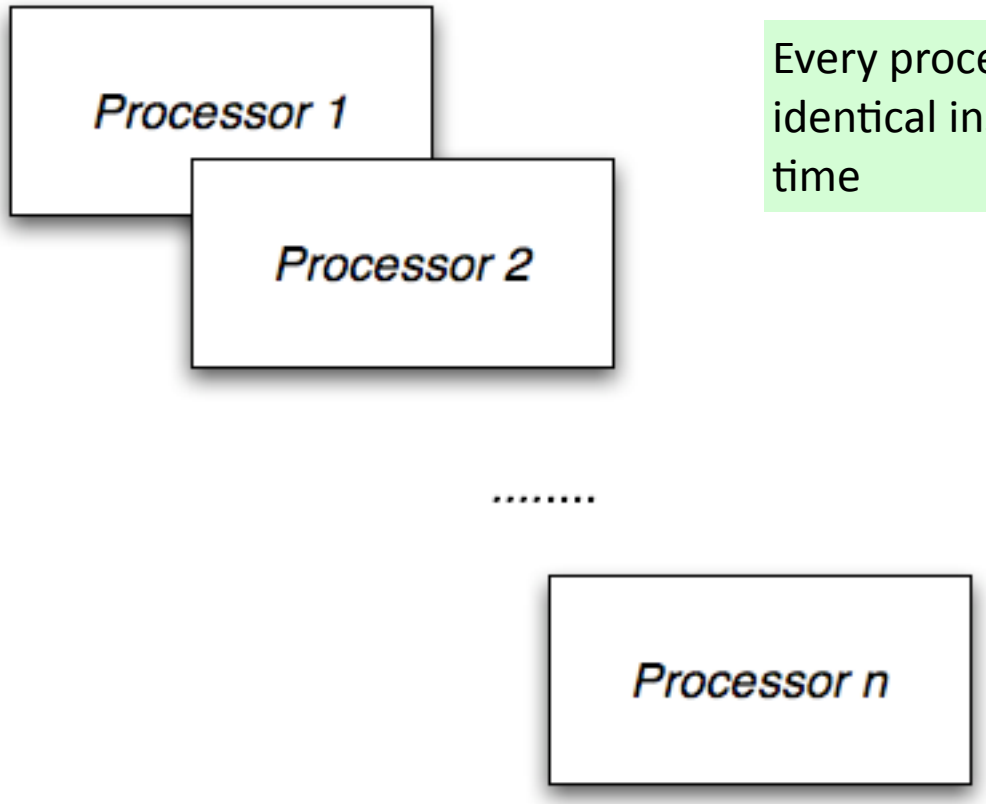
$C(3,1) = A(3,1) + B(3,1)$

.....

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

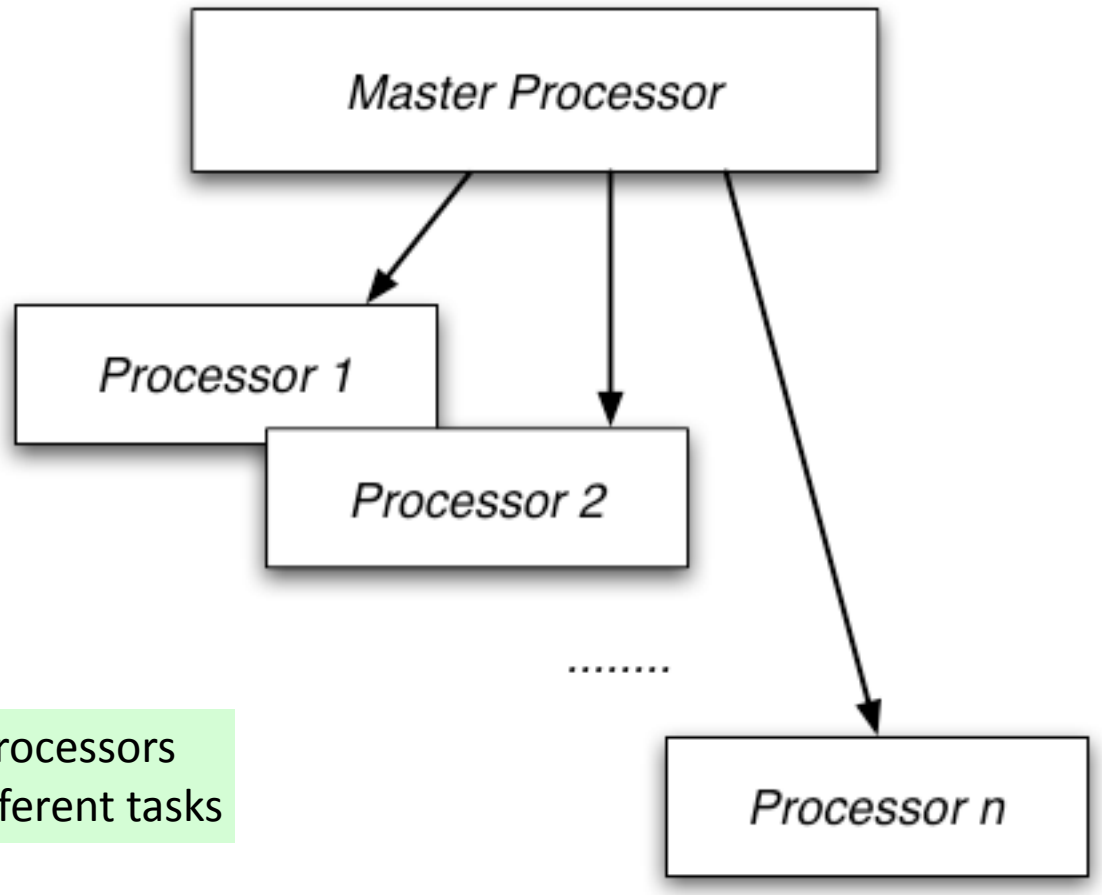
The same function is applied to all records of an input stream and a number of records can be processed simultaneously without waiting for results from previous records.

Single **I**nstruction **M**ultiple **D**ata



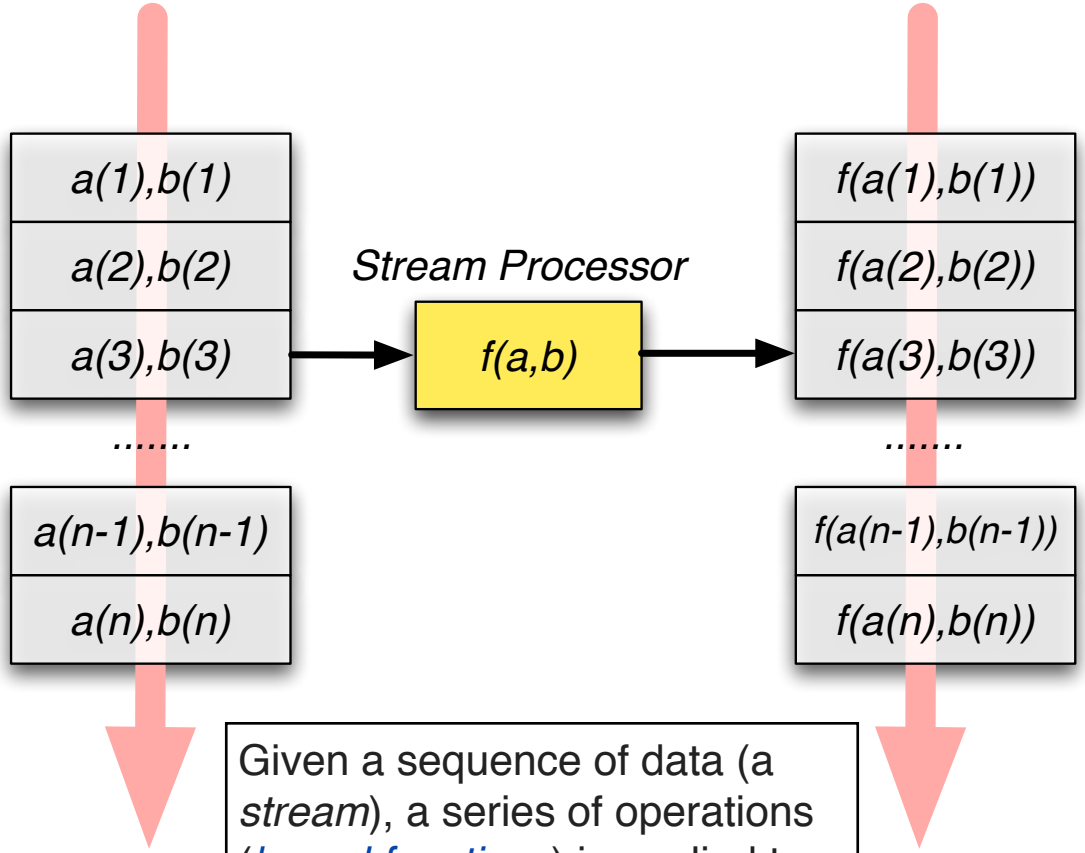
Every processor executes the identical instruction at the same time

Multiple **I**nstruction **M**ultiple **D**ata



Different processors execute different tasks

Stream Processing

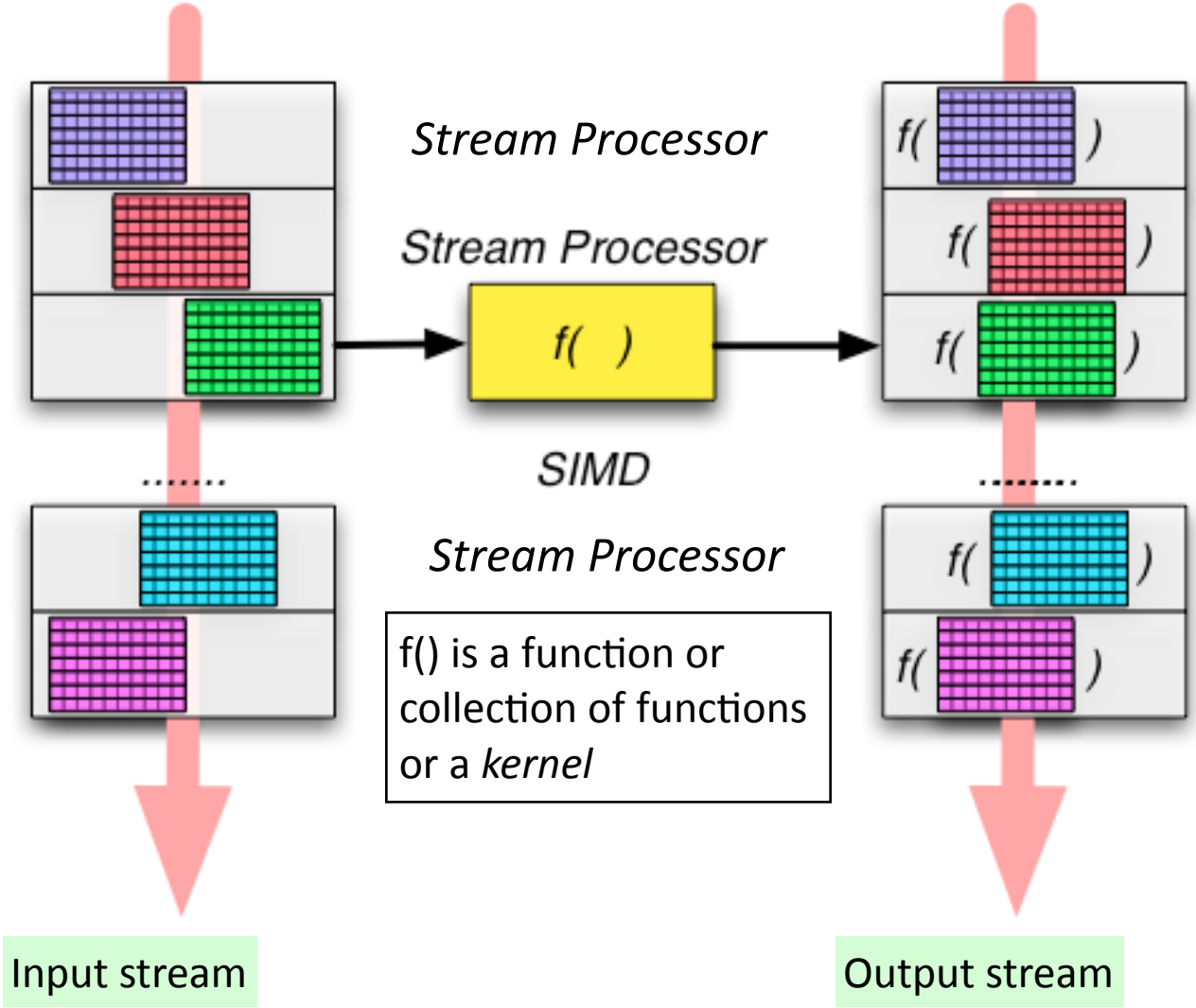


Given a sequence of data (a *stream*), a series of operations (*kernel functions*) is applied to each element in the stream.

Input stream

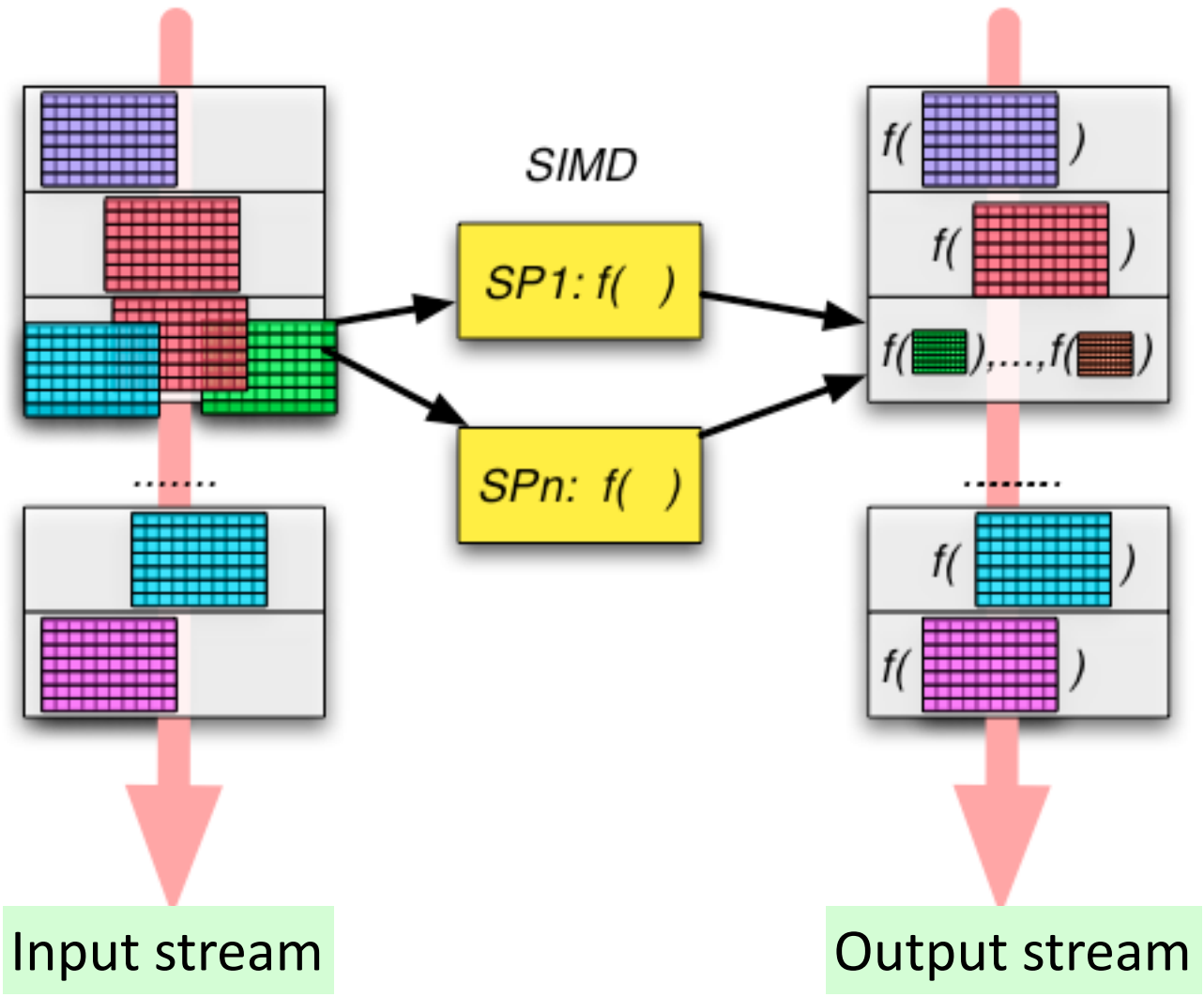
Output stream

Stream Processing
block of threads



f() is a function or collection of functions or a *kernel*

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

Current version in 2017: CUDA 8.5
No new languages (on the Horizon). CUDA and OpenCl dominate

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

I am not on top of developments since 2010

Deficiencies of OpenCL

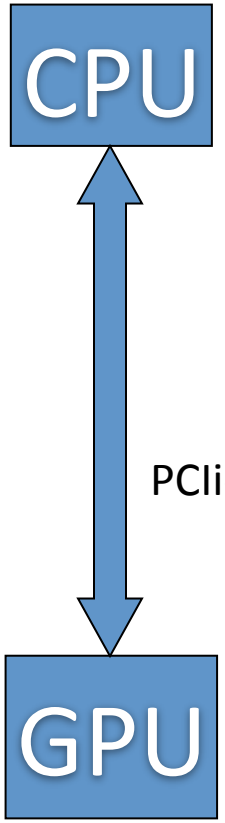
- For the most part, OpenCL maps nicely to CUDA and vice-versa. However,
- No templates, namespaces (partial support in 2013)
  - Difficult to create general frameworks
- Limitations on function arguments
- Constant parameters are hard to manage
- Hard to manage pointers
- No structures of structures 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

OpenCL vs CUDA on Pascal

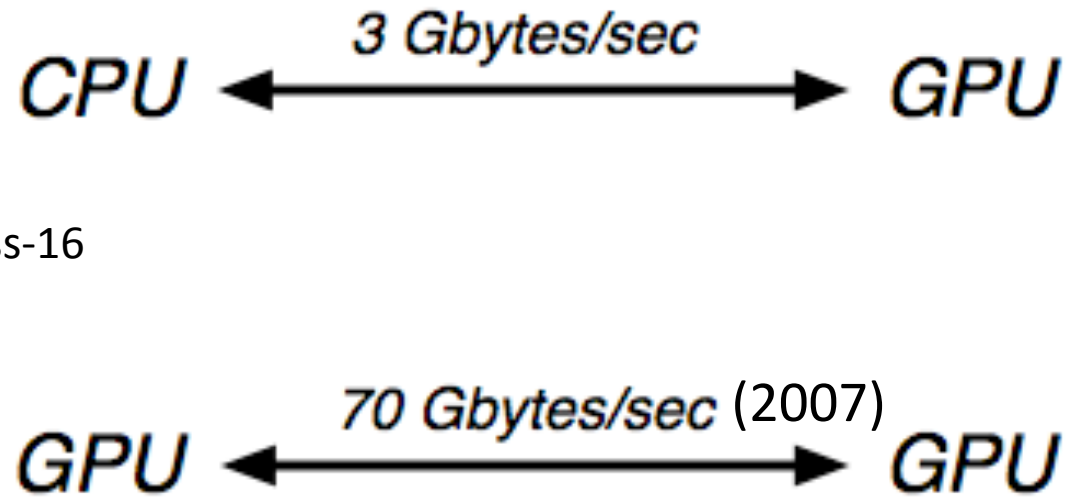
- Micro-benchmarks: identical performance
- Triad operations or FFTs
  - OpenCL: about 50% speed of CUDA

- CUDA
  - Templates, advanced libraries (math, fits, neural networks, etc.)
- OpenCL
  - Far behind. Not as fast as CUDA on Nvidia cards.
  - Only way to take advantage of AMD Radeon cards.
- source: https://www.phoronix.com/scan.php?page=news_item&px=GTX-1080-OpenCL-vs-CUDA

Memory Bandwidth (2010)



PCIie2 Express-16



177 GBytes/sec for GTX480 (2010)
up to 700 GBytes/sec for Pascal (2017)

PCIie3 (2010), 5 Gbyte/sec

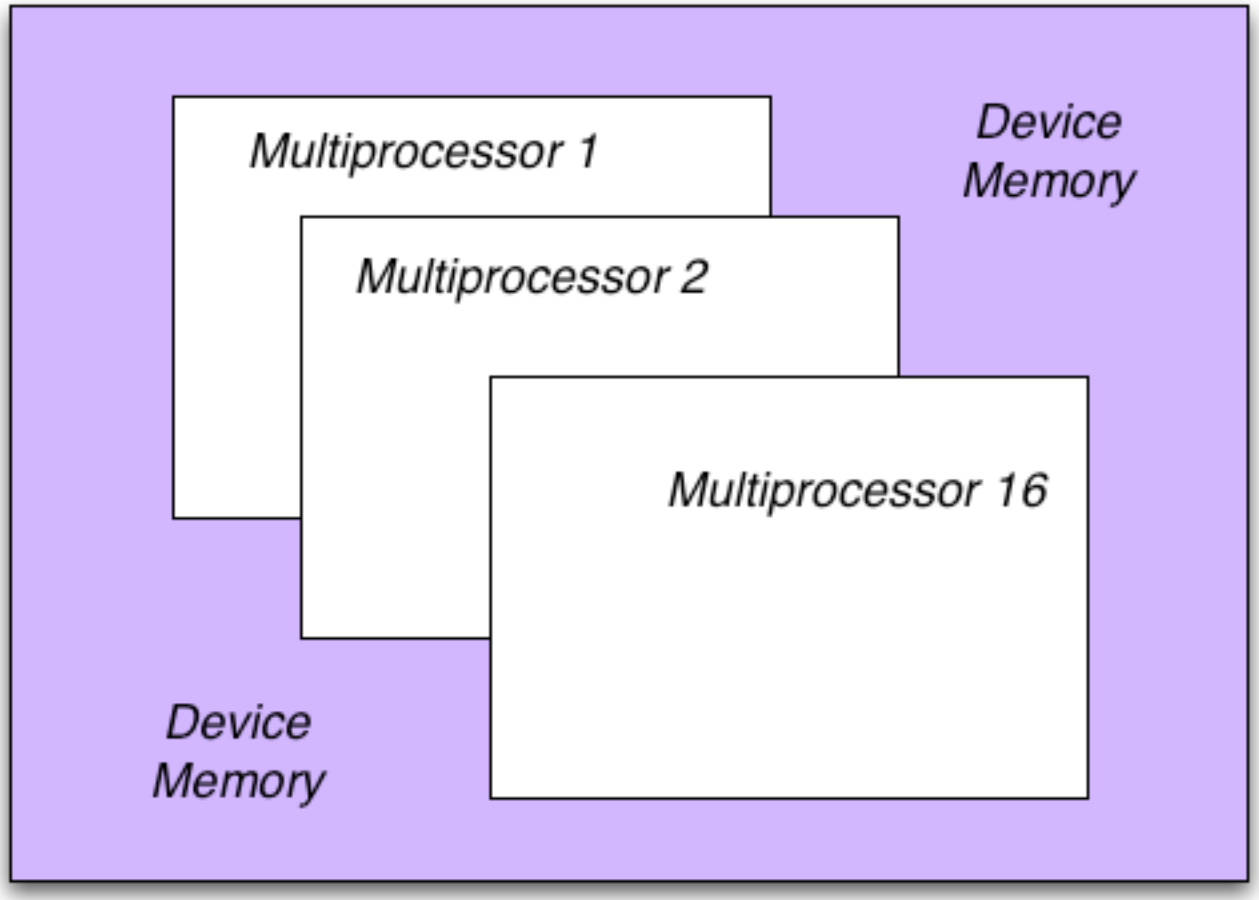
Increased disparity between speeds

GPU Architectures:

	Tesla	Fermi	Kepler	Maxwell	Pascal	
Card	GeForce 100-300	GeForce 400-500	GeForce 600-700	GeForce 900	GeForce 10	
Year	2007-2010	2010-2011	2012-2014	2014-2016	2016-2017	
Memory Bandwidth	10-200	10-300	20-600	14-300	100-500	GB/s
Single Precision	20-500	100-3000	100-8000	1000-6000	1700-10000	GFlops
Double Precision	N/A	N/A	14-2000	32-200	54-300	GFlops
Cost	N/A	$60-$500	$45-$3000	$150-$1000	$100-$700	

SOURCE: https://en.wikipedia.org/wiki/List_of_Nvidia_graphics_processing_units#GeForce_600_series

Several Streaming Multiprocessors
per GPU (280GTX has 24 MP)
480GTX has **15 SMP**
GTX1080 has **20 SMP**

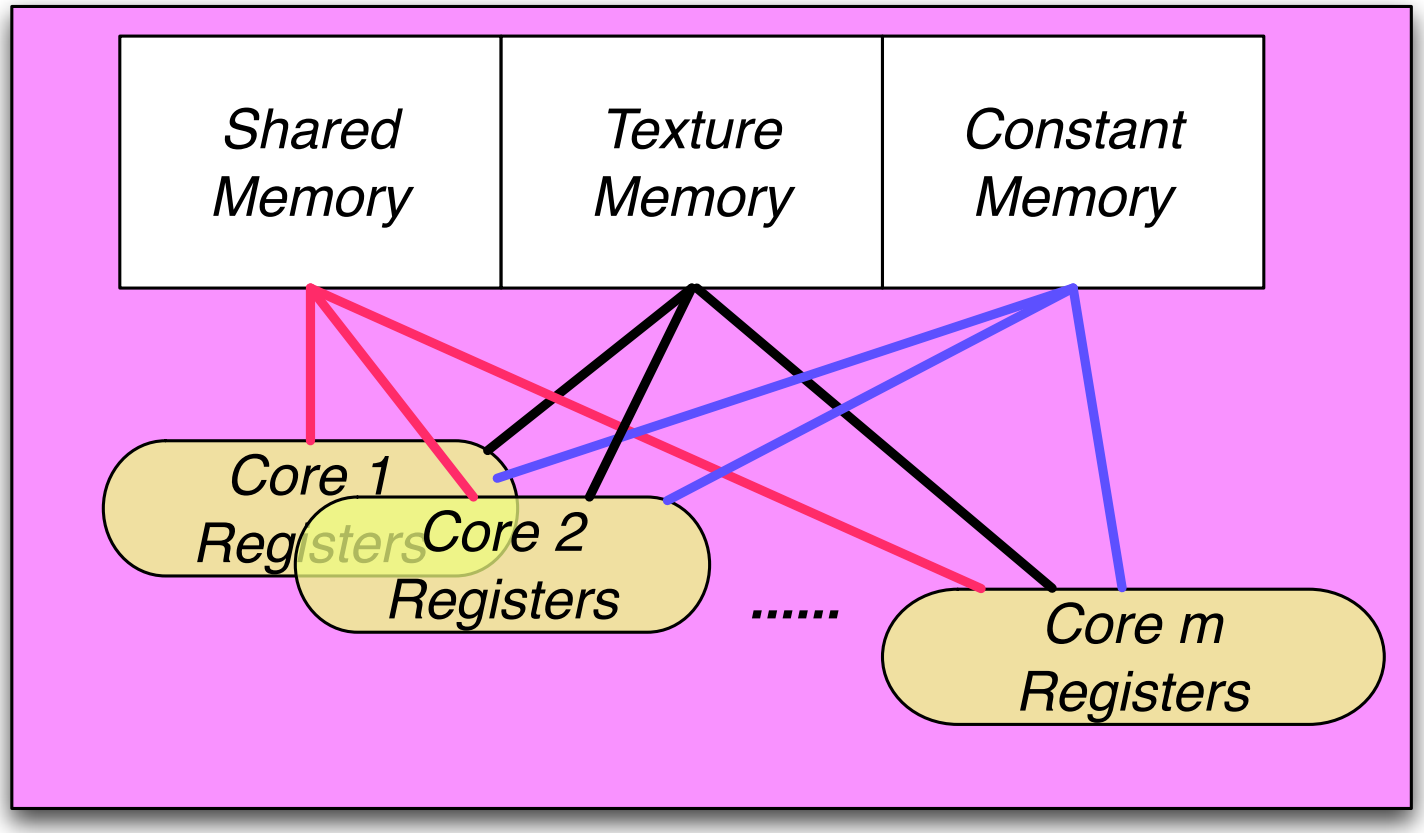


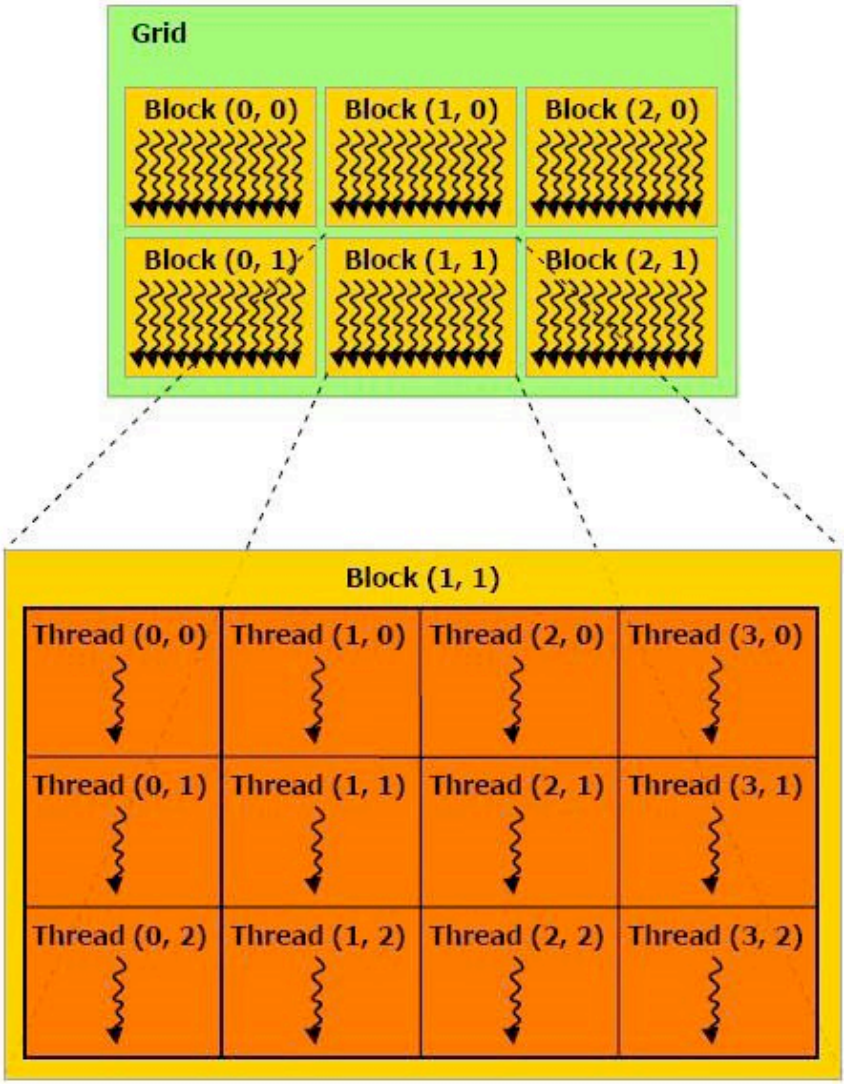
Streaming Multiprocessor

Multiple stream processors (i.e., core)

GTX480: **32 cores** per SM

GTX1080: **128 cores** per SM





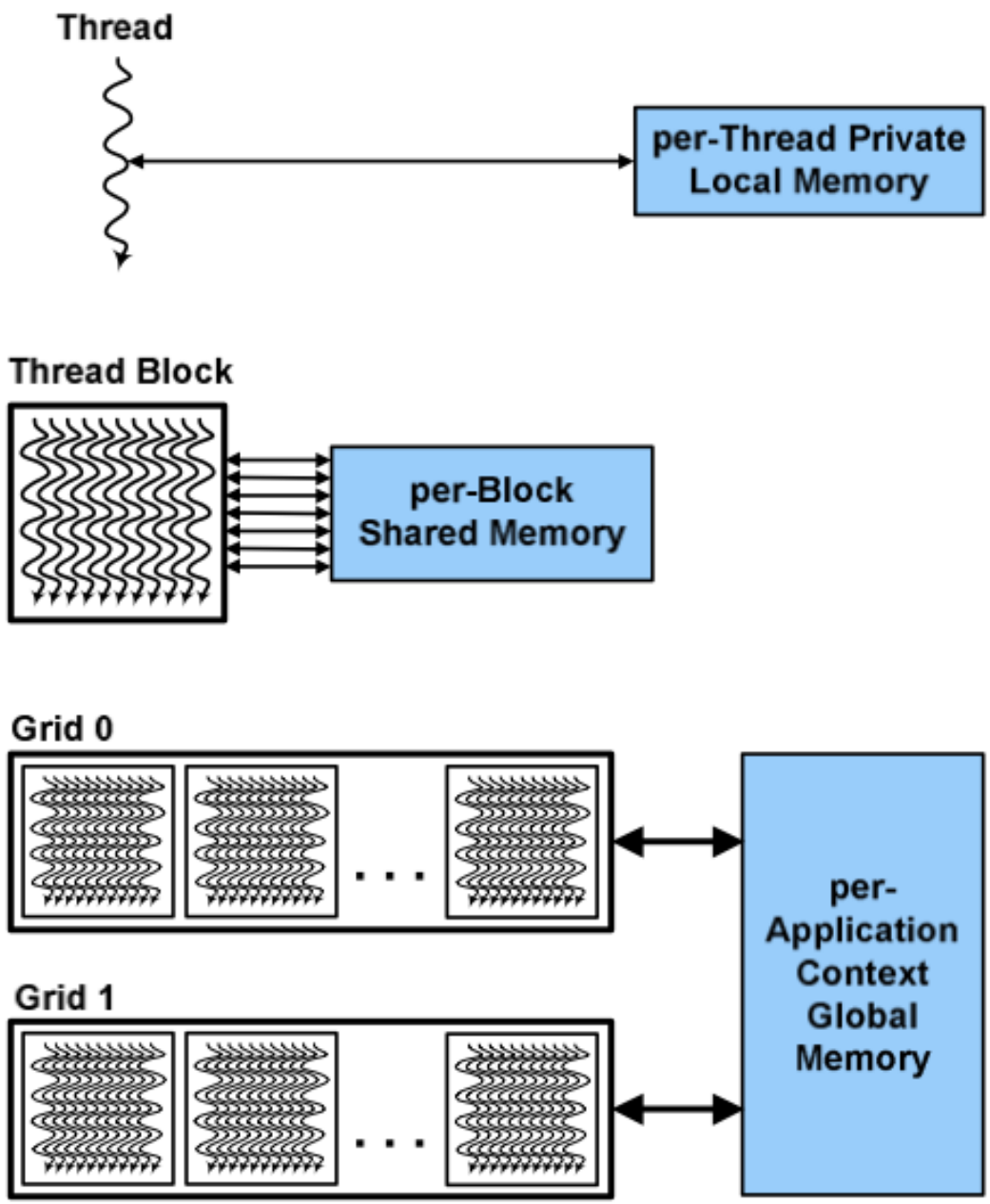
Input to GPU

Grid of Blocks

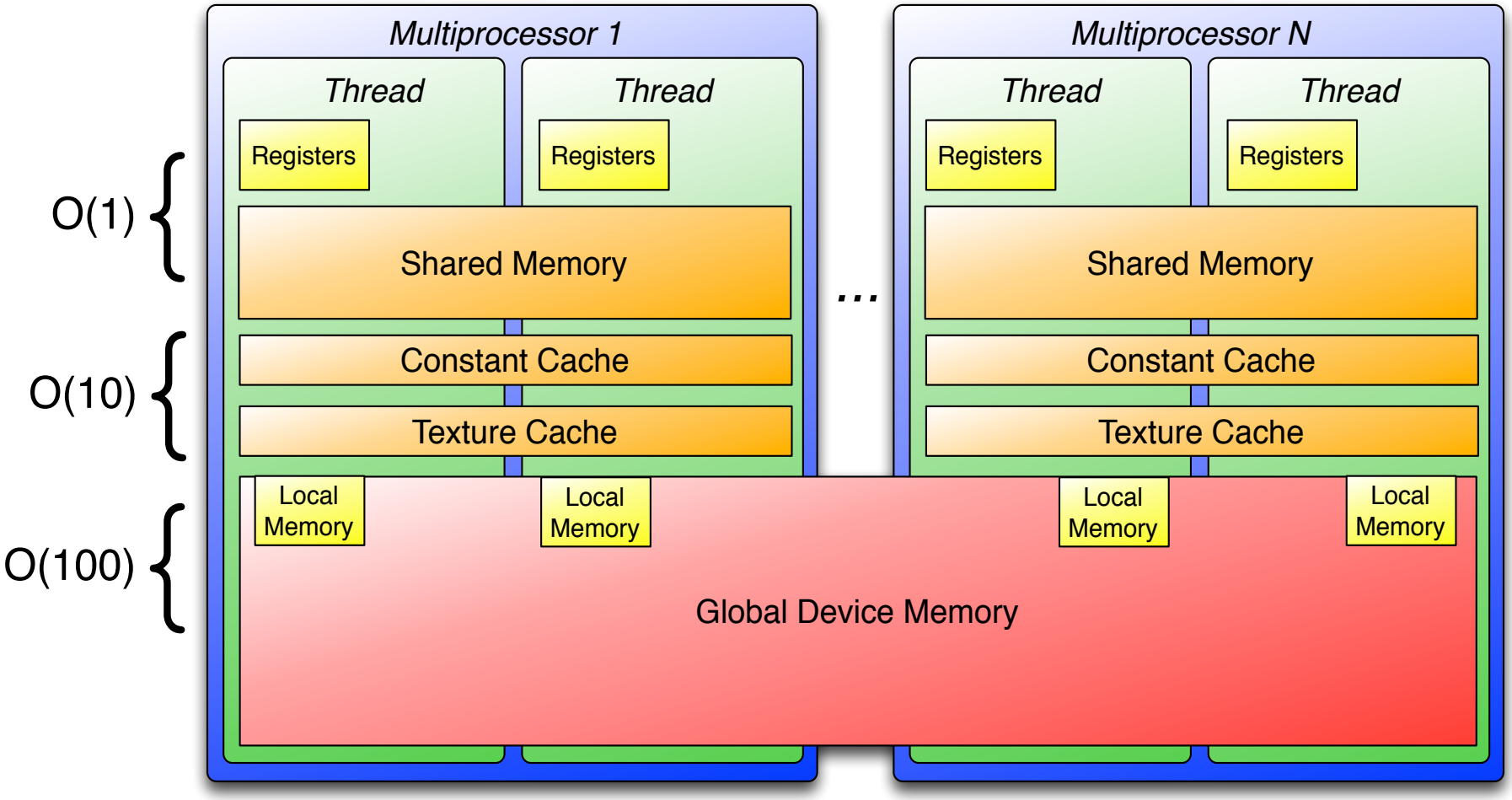
Think of a block as an array

Single Stream Processor

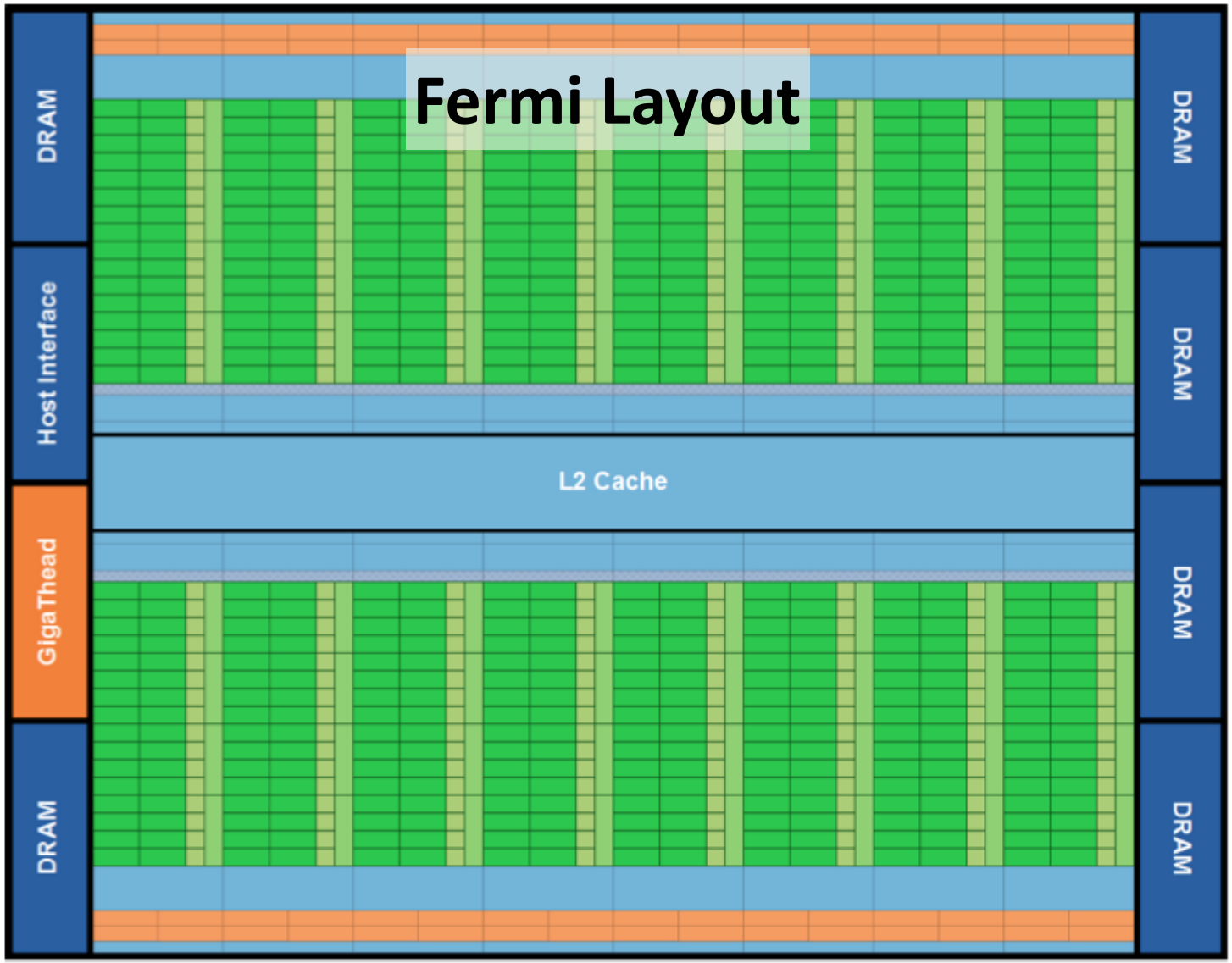
Each block is a grid of threads


Thread
per-Thread Private Local Memory
Thread Block
per-Block Shared Memory
Grid 0
Grid 1
per-Application Context Global Memory

Anatomy of Unified Architecture GEForce 8x



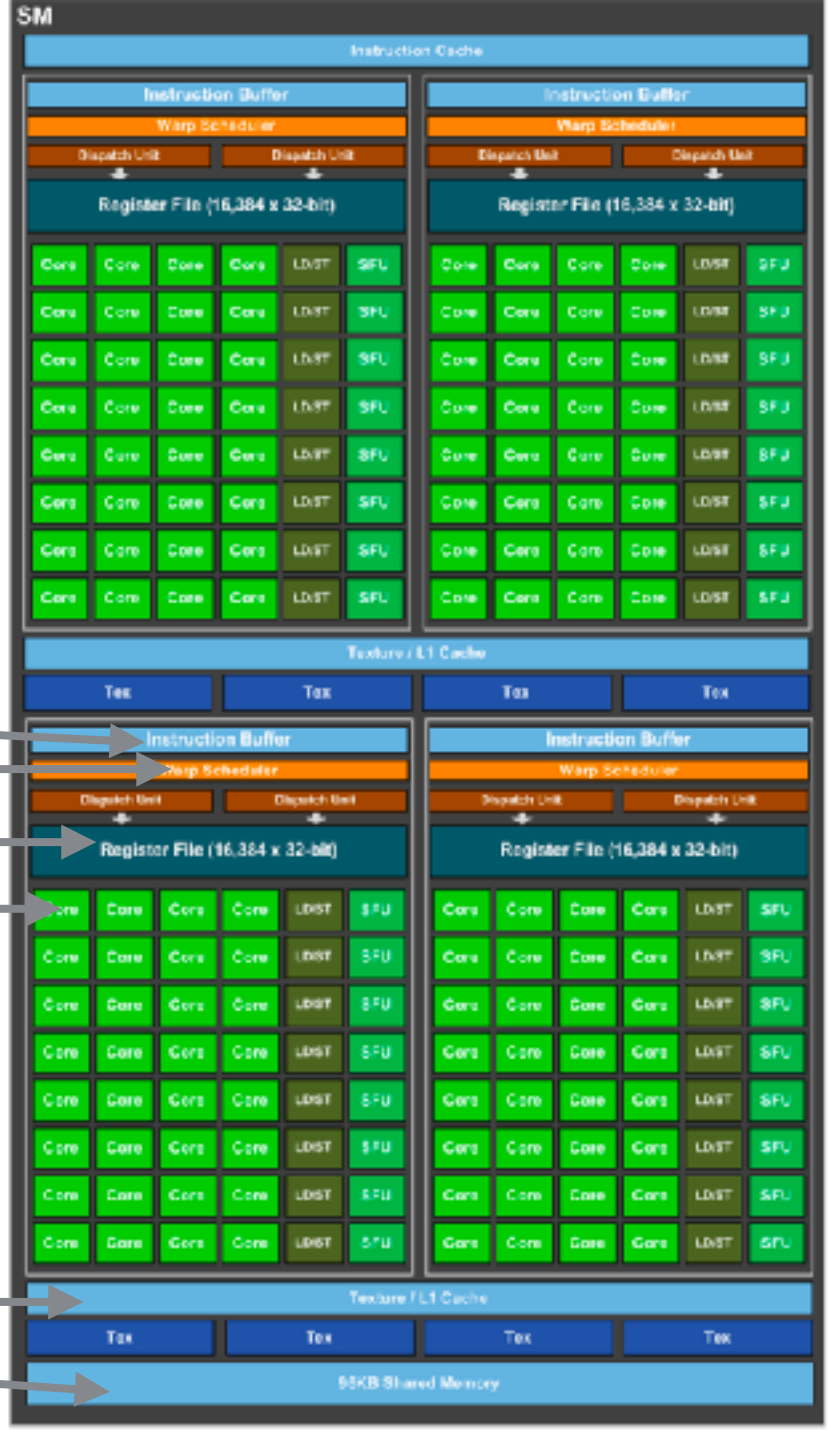
Fermi with 16 SM Processors (2010)



Fermi's 16 SM are positioned around a common L2 cache. Each SM is a vertical

Pascal GP100 with 60 SM units (2016)

Pascal Single SM Processor



Instruction Buffer

Warp scheduler

Register file

Core

Texture/L1 cache

96 kB shared memory

	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				

GPU	GeForce GTX 980 (Maxwell)	GeForce GTX 1080 (Pascal)
SMs	16	20
CUDA Cores	2048	2560
Base Clock	1126 MHz	1607 MHz
GPU Boost Clock	1216 MHz	1733 MHz
GFLOPs	4981[1]	8873[1]
Texture Units	128	160
Texel fill-rate	155.6 Gigatexels/sec	277.3 Gigatexels/sec
Memory Clock (Data Rate)	7,000 MHz	10,000 MHz
Memory Bandwidth	224 GB/sec	320 GB/sec
ROPs	64	64
L2 Cache Size	2048 KB	2048 KB
TDP	165 Watts	180 Watts
Transistors	5.2 billion	7.2 billion
Die Size	398 mm^2	314 mm^2
Manufacturing Process	28 nm	16 nm

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
- 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: 32x16 = 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
- **Lots of room for code optimization**
  - **More recent ideas:**
  - Maximize register usage
  - Can achieve close to peak performance with subset of

In 2017

- Unified Memory
- Instruction Preemption
- Multiple kernels per GPU
- GPU to GPU communication via NVlink

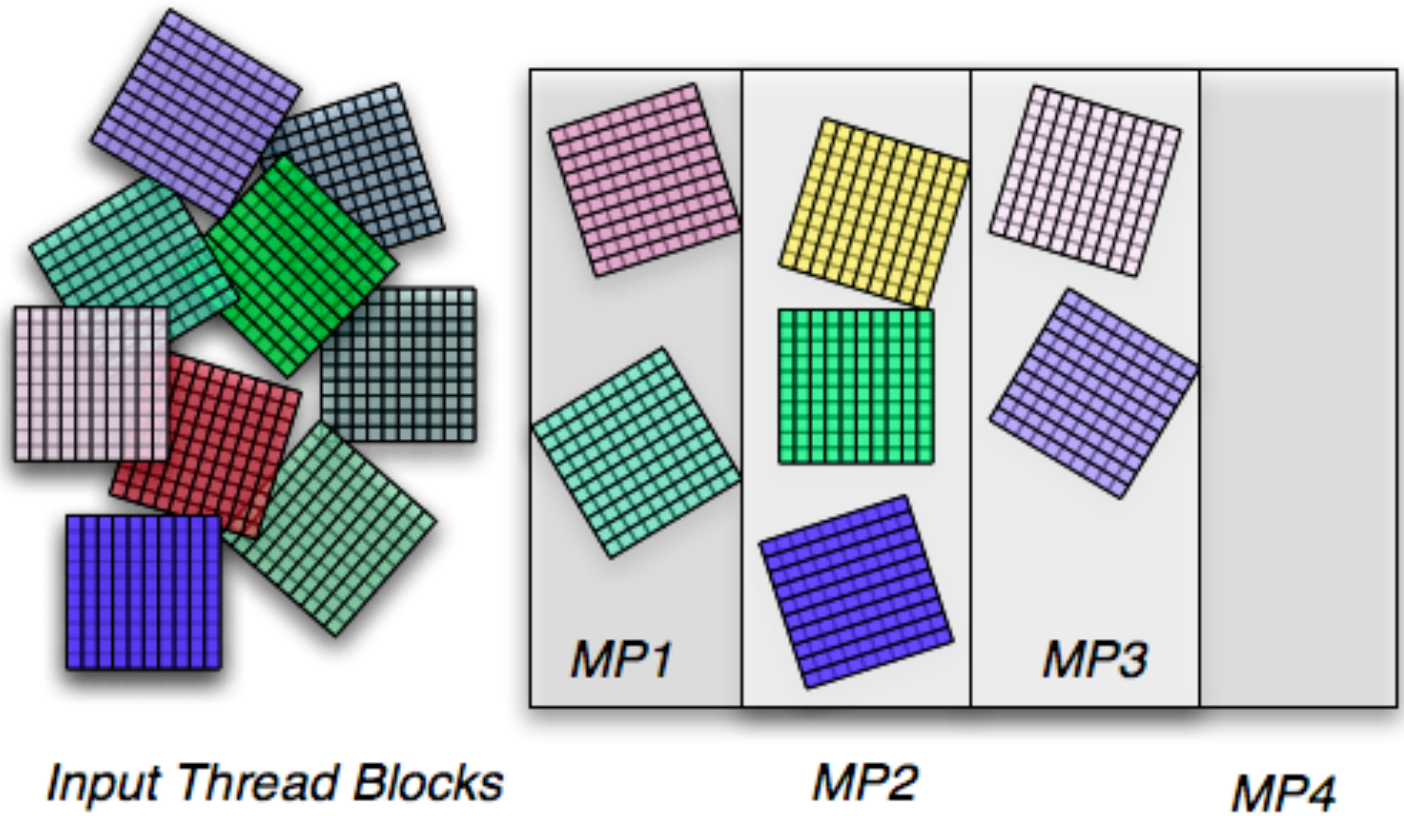
- All the above ease programming
- Increase efficiency with less effort
- Closer to unifying CPU and GPU

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 ←→ CPU (less important in 2017)

  Minimize transfers

  Device Memory ←→ GPU registers

  More automated with unified memory
- 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

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?

Two Fundamental Rules (in 2010)
(I have not studied this for Pascal architecture)

- 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)

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 (96 Kbytes on GTX1080) of shared memory per SM
  - Under control of the user
  - Be frugal

Unified Memory

- Single virtual address space across CPUs and GPUs
- Decreases need to explicitly manage memory allocation on multiple devices and manage data transfer and synchronization

Unified Memory History

- Fermi Architecture (2009)
  - Unified GPU address space across (private local memory, thread block shared memory, global memory)
  - Allowed for full pointer support in C/C++
    - a single pointer could point to any of the three memory blocks
- CUDA 4.0 (2011)
  - Unified Virtual Addressing (UVA)
  - Single virtual space across CPU and GPU
  - Allows for Zero-copy memory (pinned CPU memory accessible by GPU directly) over PCIe. Useful, but non-performing due to inefficiencies of PCIe (low bandwidth, high latency)

Unified Memory History

- CUDA 6.0
  - pool of managed memory shared between CPU and GPU through a single pointer
  - data migration is handled automatically
  - code on the CPU and the GPU use the same pointer
  - Caveats: Limitations of Kepler and Maxwell architectures
    - managed memory required synchronization before kernel launch
    - CPU and GPU could not *simultaneously* access this memory
    - Memory size is bound by the amount of GPU memory

Example source code

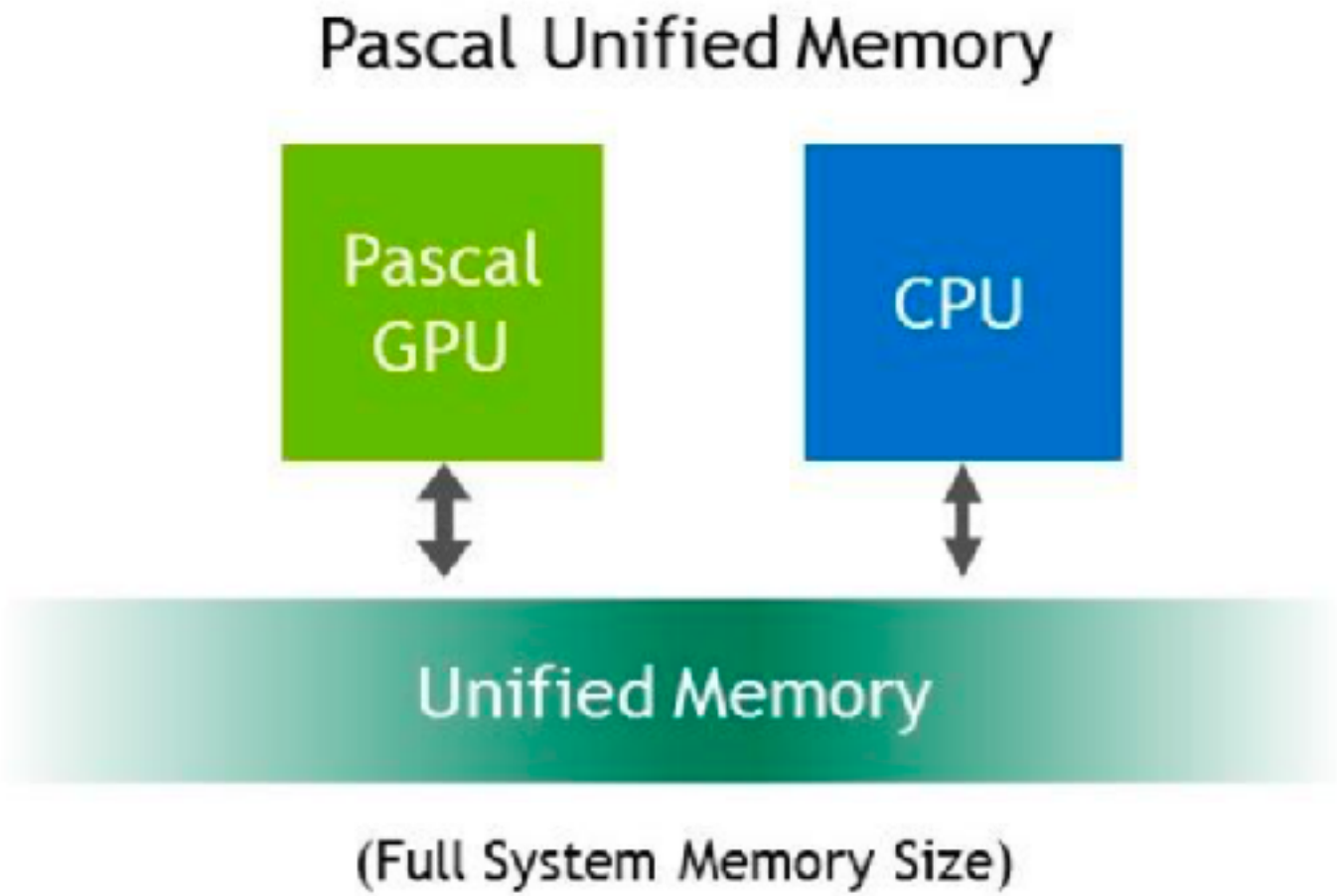
CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

CUDA 6 Code with Unified Memory

```
void sortfile(FILE *fp, int N) {
  char *data;
  cudaMallocManaged(&data, N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  cudaFree(data);
}
```

Unified Memory in Pascal



Pascal Unified Memory

CPU Code

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort(data, N, 1, compare);


  use_data(data);

  free(data);
}
```

Pascal Unified Memory*

```
void sortfile(FILE *fp, int N) {
  char *data;
  data = (char *)malloc(N);

  fread(data, 1, N, fp);

  qsort<<<...>>>(data,N,1,compare);
  cudaDeviceSynchronize();

  use_data(data);

  free(data);
}
```

*with operating system support

malloc(N) can allocate memory accessible from any CPU and GPU in the system

Deep Learning: DGX-1

NVIDIA DGX-1 System Specifications

Specification	Value
GPUs	8x Tesla P100 GPUs
TFLOPS	170 (GPU FP16) + 3 (CPU FP32)
GPU Memory	16 GB per GPU / 128 GB per DGX-1 Node
CPU	Dual 20-core Intel® Xeon® E5-2698 v4 2.2 GHz
NVIDIA CUDA Cores	28,672
System Memory	512 GB 2133 MHz DDR4 LRDIMM
Storage	4x 1.92TB SSD RAID 0
Network	Dual 10 GbE, 4 IB EDR
System Weight	134 lbs
System Dimensions	866 D x 444 W x 131 H (mm)
Packing Dimensions	1180 D x 730 W x 284 H (mm)
Power	3200W (Max). Four 1600W load-balancing power supplies (3+1 redundant), 200-240V (ac), 10A
Operating Temperature Range	10 - 35°C

Resources

- GPU Programming Guide 3.1
  - http://www.nvidia.com/
- Better Performance at Lower Occupancy
  - By Vasily Volvov, 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

The Transpose

- Brute force to super-efficient
- Describe the journey

- Naïve implementation
- Global memory to swap operation
- Global memory to shared memory to swap operations
- Remove bank conflicts
- Partition camping

What is a transpose?

- Given a matrix A(n,m), calculate
  - B(n,m) = A(m,n)
  - $B = A^T$

A =

3	5	7	12.5
-34	32	2	-3

= 2 x 4 matrix

B =

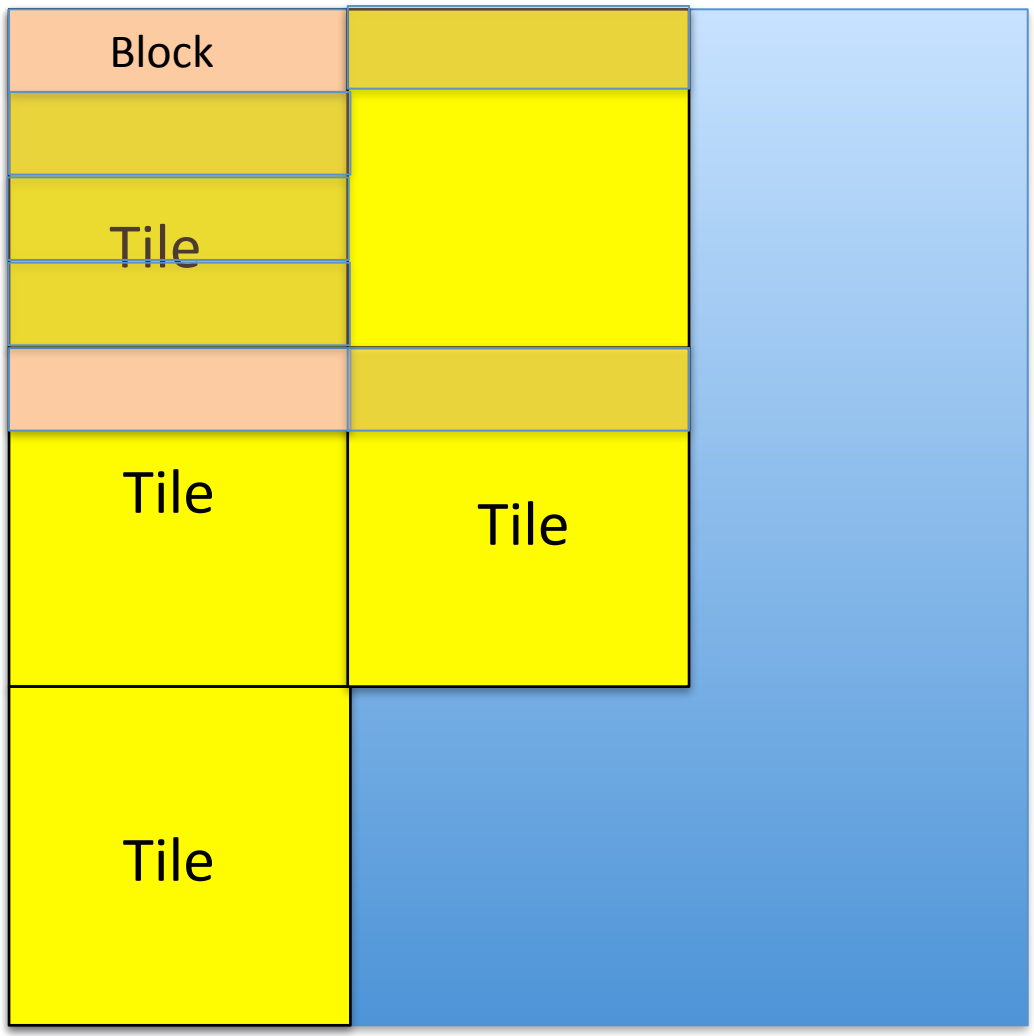
3	-34
5	32
7	2
12.5	-3

= 4 x 2 matrix

Thread, Block, Grid

- Block: 8 x 32 = 256 threads
  - blockDim.x X blockDim.y
- Array size: 2048 x 2048
  - Divided into tiles of size 32 x 32
- Each block handles a single tile
- Grid of blocks has size:
  - gridDim.x X gridDim.y
  - (2048/8) x (2048/32) = 256 x 64 (grid size)

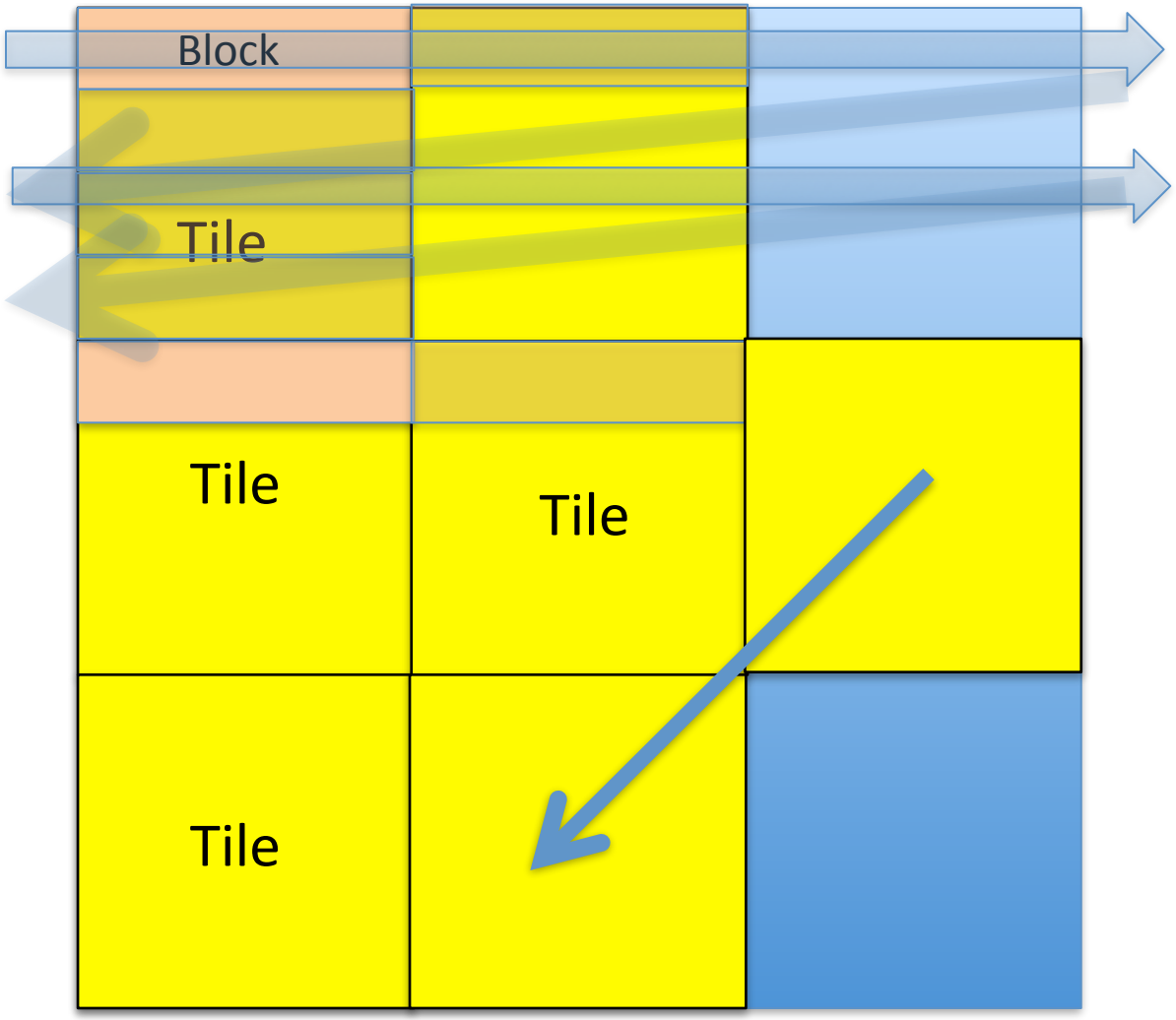
Array layout



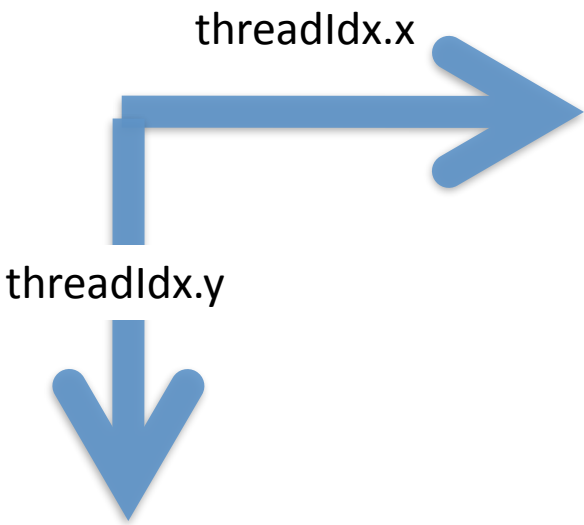
Block: 8 x 32
Tile: 32 x 32
Array: 2048 x 2048

blockIdx.x: block# along x
blockIdx.y: block# along y

Memory Layout



Block: 8 x 32
Tile: 32 x 32
Array: 2048 x 2048



Simple Copy

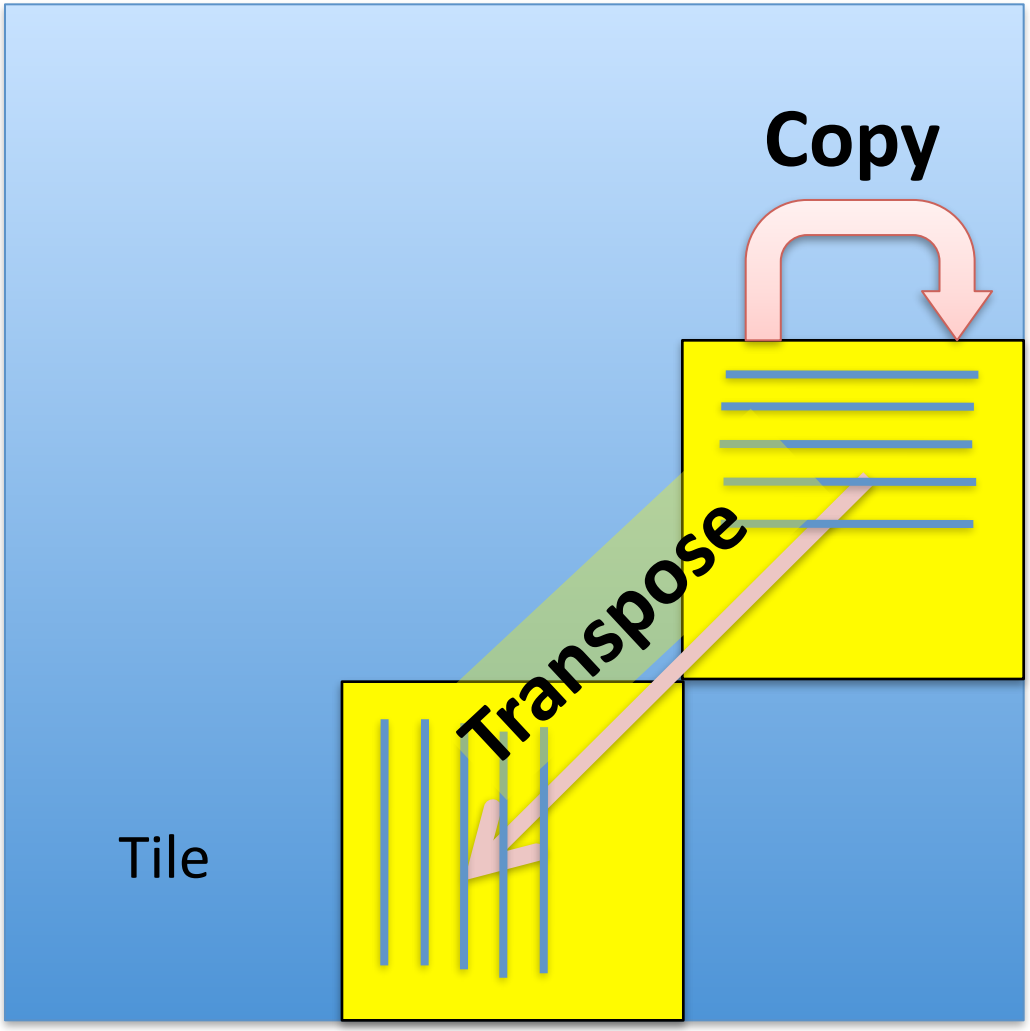
```
__global__ void copy(float *odata, float* idata, int width,
                                    int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

  int index  = xIndex + width*yIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index+i*width] = idata[index+i*width];
    }
  }
}
```

i = 0, 1, 2, 3: subset of tile to be filed by 8.x32 block of threads

Each thread is responsible for four swaps

2D index (xindex, yindex) converted to 1D index (index)

Non-coalesced transfers



Naïve Transpose

```
__global__ void transposeNaive(float *odata, float* idata,
                               int width, int height, int nreps)
{
  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

  int index_in  = xIndex + width * yIndex;
  int index_out = yIndex + height * xIndex;
  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i] = idata[index_in+i*width];
    }
  }
}
```

Same structure as copy, EXCEPT
index_out and index_in are different

threadIdx.x += 1
index_out += height

Timings

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2

Extremely poor!!!

Problem: non-coalesced transfers

- Compute capability 1.3 (GTX280)
  - Memory is divided into 32, 64, or 128 byte segments (depending on the word size)
  - Complex, but the following is sufficient (not necessary) for optimal (coalesced) transfers for floats (4 byte words)
  - All threads must have different addresses and lie in the same 64 byte segment
- Compute capability 2.0 (GTX480)
  - Access to global memory go through L2 cache
  - Non-coaslesced is of “almost” no concern
  - It is replaced by cache reuse

Shared Memory

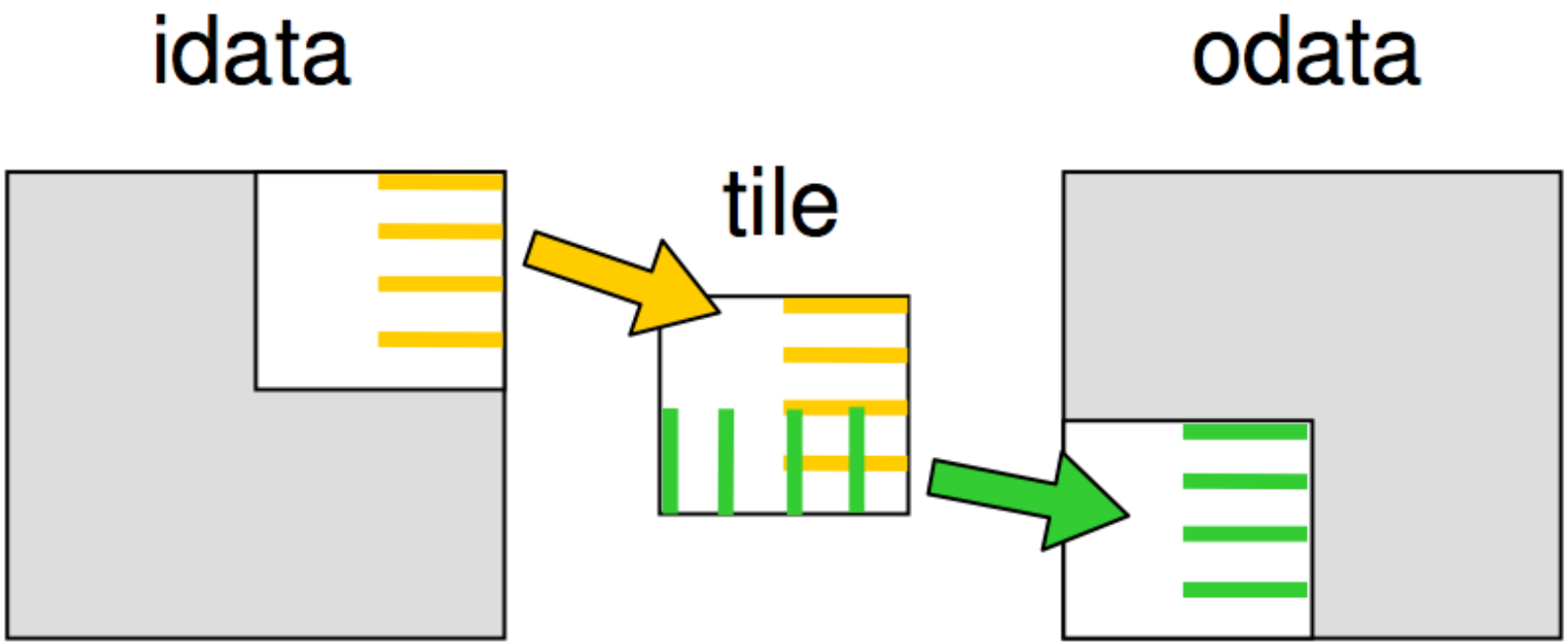
- Compute Capability 1.x (GTX280 and below)
  - 16 banks
  - All threads of half-warp must access different addresses
- Compute Capability 2.0 (GTX480, Fermi)
  - 32 banks
  - All threads of full warp must access different addresses

Use of shared memory

Input array ➔ shared memory

Shared memory ➔ output array

Write out odata in coalesced form



```
__global__ void transposeCoalesced(float *odata,
            float *idata, int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM];

  int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r < nreps; r++) {
    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      tile[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
      odata[index_out+i*height] =
        tile[threadIdx.x][threadIdx.y+i];
    }
  }
}
```

Shared Memory

Coalesced write

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

Speedup of 8x

Still 6x slower than simple copy!

Shared Memory Copy

```
__shared__ float tile[TILE_DIM][TILE_DIM];

int xIndex = blockIdx.x*TILE_DIM + threadIdx.x;
int yIndex = blockIdx.y*TILE_DIM + threadIdx.y;

int index  = xIndex + width*yIndex;
for (int r=0; r < nreps; r++) {
  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    tile[threadIdx.y+i][threadIdx.x] =
      idata[index+i*width];
  }

  __syncthreads();

  for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS) {
    odata[index+i*width] =
      tile[threadIdx.y+i][threadIdx.x];
  }
}
```

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1

Time did not change

Shared memory not useful in copy

But keep copy as close as possible to transpose

Bank Conflicts

- Reading rows of 32x32 tile
  - No bank conflicts
- Writing columns of 32x32 tile
  - Entire column lies in the same bank
  - Bank conflict! Strong serialization
- To avoid bank conflict
  - Add an addition column to the tile
    - Tile[32][33]
    - Tile[i][j] and Tile[i+1][j] are separated by 33 elements, so lie in different banks! No more bank conflicts

Remove bank conflicts: CODE

```
__global__ void transposeFineGrained(float *odata,
            float *idata, int width, int height,  int nreps)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index = xIndex + (yIndex)*width;
```

0,	1, 2, …, 31,	
32,	33,	…, 63,
64,	65,	…, 95,

Block[TILE_DIM][TILE_DIM]

Bank conflicts

0,	1, 2, …, 32	
33,	34,	…, 65,
66,	67,	…, 98,

Block[TILE_DIM][TILE_DIM+1]

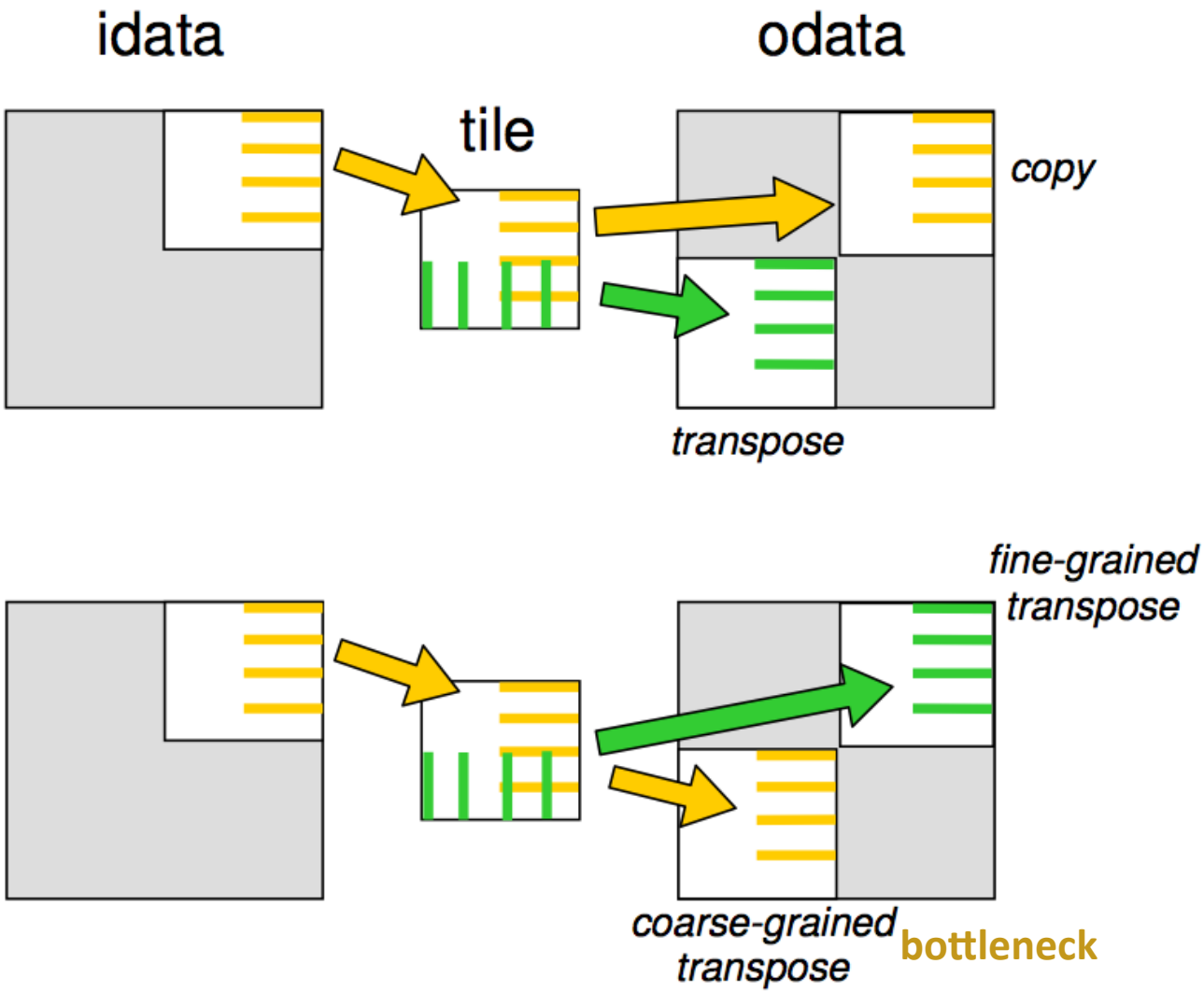
No Bank conflicts

Remove Bank Conflicts

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2

In this case, bank conflicts had no impact.
But it is better not to have them.

What is the Problem?



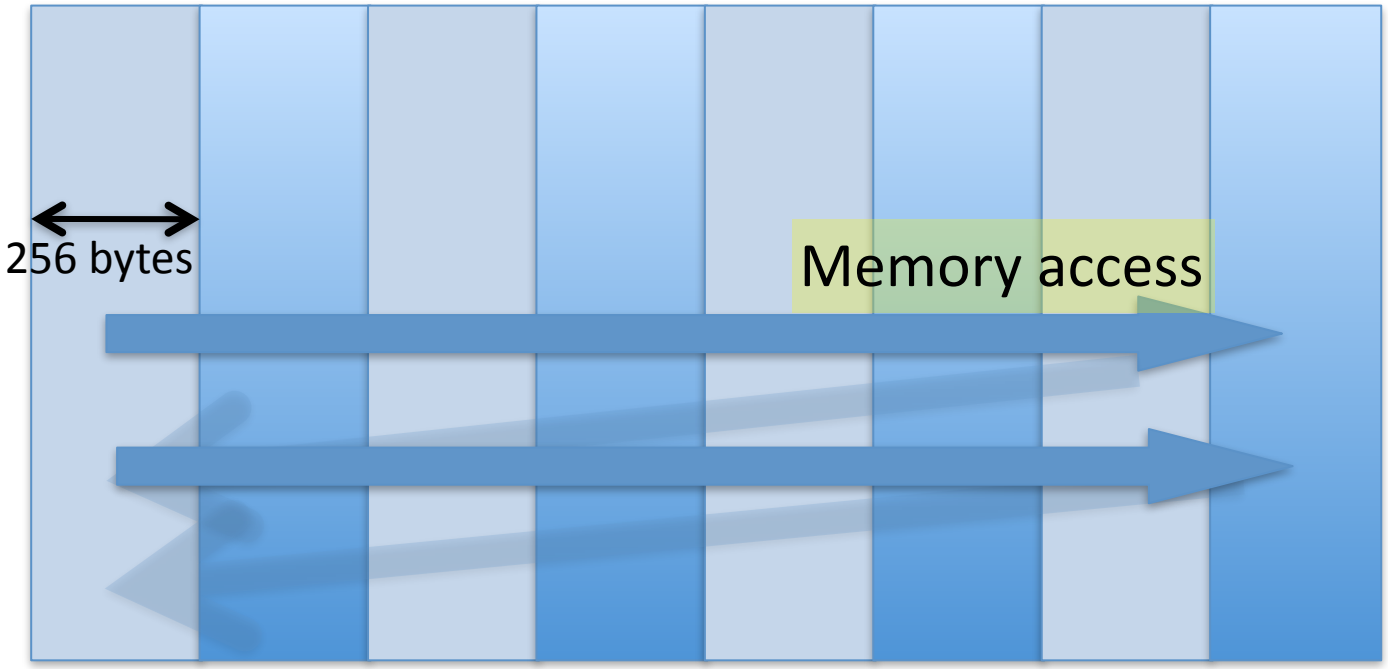
```
__global__ void transposeCoarseGrained(float *odata,
      float *idata, int width, int height, int nreps)
{
  __shared__ float block[TILE_DIM][TILE_DIM+1];

  int xIndex = blockIdx.x * TILE_DIM + threadIdx.x;
  int yIndex = blockIdx.y * TILE_DIM + threadIdx.y;
  int index_in = xIndex + (yIndex)*width;

  xIndex = blockIdx.y * TILE_DIM + threadIdx.x;
  yIndex = blockIdx.x * TILE_DIM + threadIdx.y;
  int index_out = xIndex + (yIndex)*height;

  for (int r=0; r<nreps; r++) {
    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      block[threadIdx.y+i][threadIdx.x] =
        idata[index_in+i*width];
    }

    __syncthreads();

    for (int i=0; i<TILE_DIM; i += BLOCK_ROWS) {
      odata[index_out+i*height] =
        block[threadIdx.y+i][threadIdx.x];
    }
  }
}
```

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*

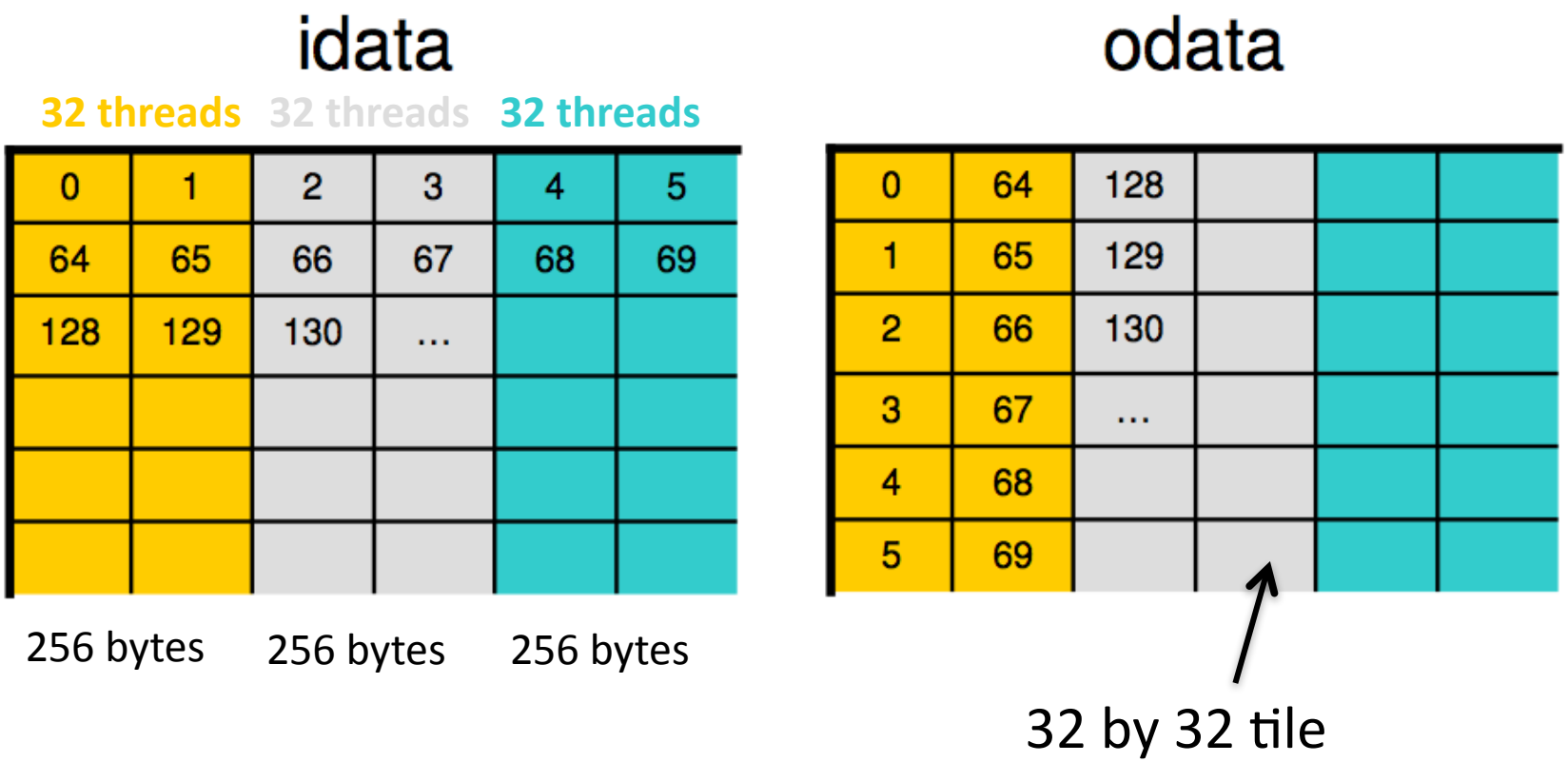
Depending on the tile written to, bandwidth goes from 16 to 80 GB/sec!!

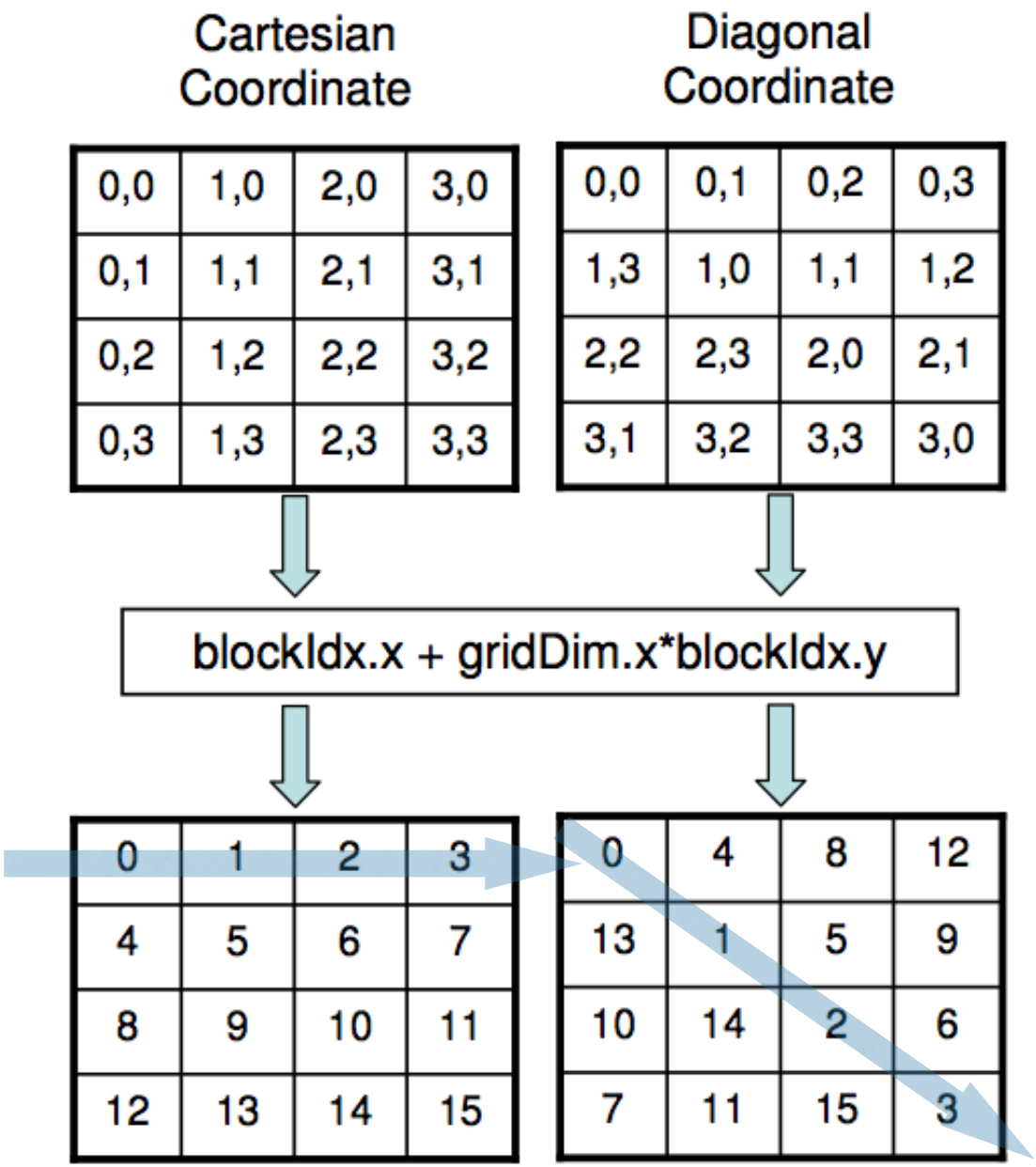
Partition Camping

- Global memory is divided into 8 partitions, each 256 bytes wide



Tile Access Patterns





x: distance along diagonal
y: different diagonal slices

Block Re-Ordering

```
__global__ void transposeDiagonal(float *odata,
            float *idata, int width, int height, int nreps)
{
  __shared__ float tile[TILE_DIM][TILE_DIM+1];

  int blockIdx_x, blockIdx_y;

  // diagonal reordering
  if (width == height) {
    blockIdx_y = blockIdx.x;
    blockIdx_x = (blockIdx.x+blockIdx.y)%gridDim.x;
  } else {
    int bid = blockIdx.x + gridDim.x*blockIdx.y;
    blockIdx_y = bid%gridDim.y;
    blockIdx_x = ((bid/gridDim.y)+blockIdx_y)%gridDim.x;
  }
```

Change block coordinate system

Replace blockIdx.x and blockIdx.y by blockIdx_x and blockIdx_y everywhere in the code

Cartesian

idata

0	1	2	3	4	5
64	65	66	67	68	69
128	129	130	…		

odata

0	64	128			
1	65	129			
2	66	130			
3	67	…			
4	68				
5	69				

Diagonal

0	64	128			
	1	65	129		
		2	66	130	
			3	67	…
				4	68
					5

0					
64	1				
128	65	2			
	129	66	3		
		130	67	4	
			…	68	5

	Effective Bandwidth (GB/s) 2048x2048, GTX 280	
	Loop over kernel	Loop in kernel
Simple Copy	96.9	81.6
Shared Memory Copy	80.9	81.1
Naïve Transpose	2.2	2.2
Coalesced Transpose	16.5	17.1
Bank Conflict Free Transpose	16.6	17.2
Fine-grained Transpose	*80.4*	*81.5*
Coarse-grained Transpose	*16.7*	*17.1*
Diagonal	69.5	78.3

Maximum achievable performance!!

GTX480

	GTX280	GTX480	GTX280	GTX480
	Throughput	Throughput	Time (ms)	Time (ms)
Simple copy	109	386	285	25
Transpose: shared copy	82	209	379	47
Transpose: naive	2.7	11	11475	877
Transpose: coalesced	21	69	1505	143
Transpose: no bank conflict	21	212	1504	46
Transpose: Diagonal	89	353	352	28

Throughput*time = constant

GTX480

	Block 32x32	Block: 8x32
	Throughput	Throughput
Simple copy	358	386
Transpose: shared copy	141	209
Transpose: naive	33	11
Transpose: coalesced	51	69
Transpose: no bank conflict	260	212
Transpose: Diagonal	289	353

8x32
Single thread does
four times more work
Less threads

Max nb blocks:
Each block: 1024 threads
or 256 threads

32x32: 1 block max
8x32 : 6 blocks max
(possibly restricted by
Shared memory

Shared memory: 48k
Used shared: 4k, enough
For 12 blocks

CUDA

Let us return to the Spectral Finite-Element code