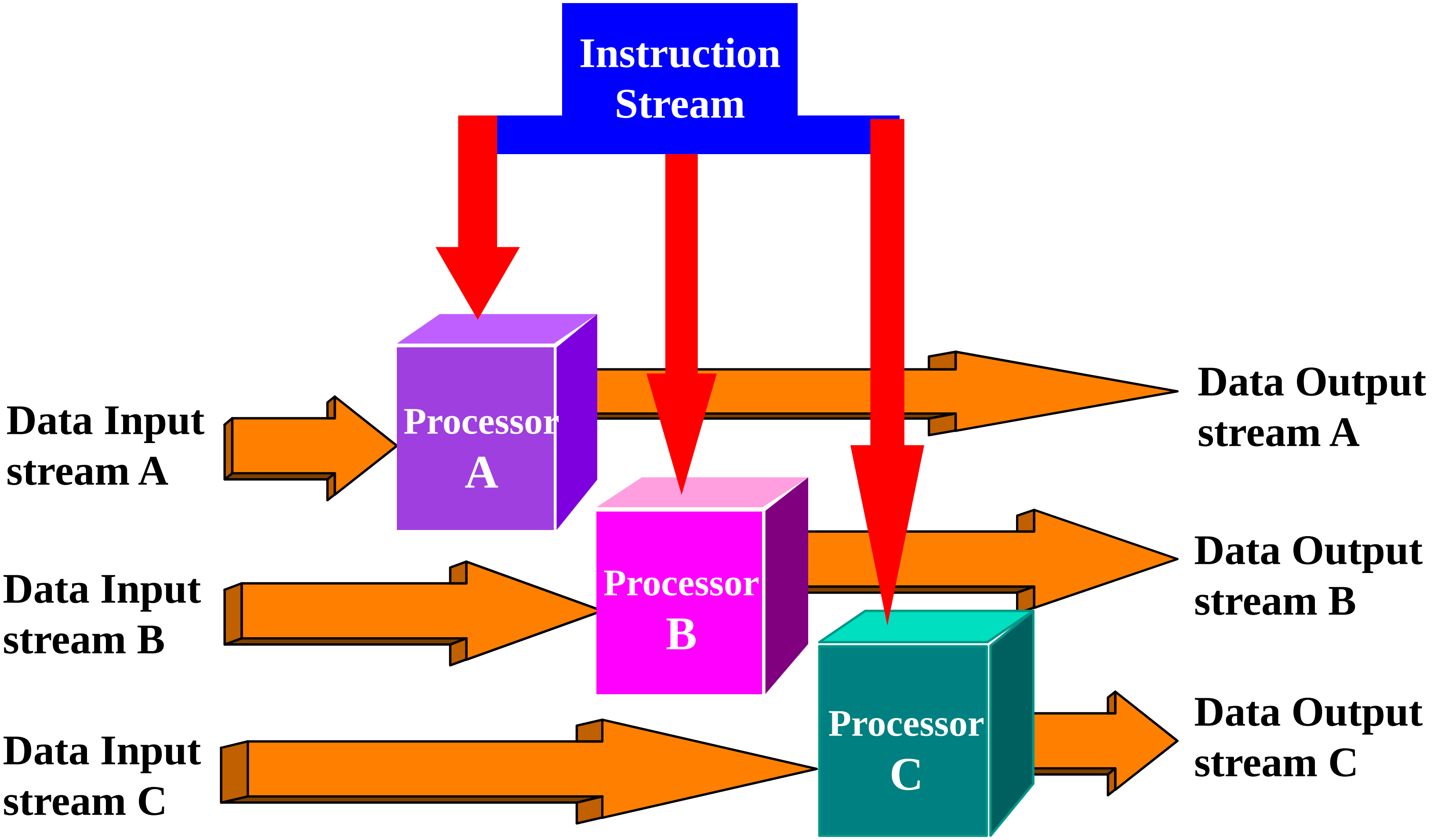
EECS 573
Microarchitecture

Data Parallel Architectures: GPUs

Todd Austin
CSE
University of Michigan
Fall 2014

With slides by David Kirk, Wen-mei W. Hwu, Li-Shiuan Peh, Mehrzad Samadi, Amir Hormati, Janghaeng Lee, and Anoushe Jamshidi

Data Parallel Architecture Recap



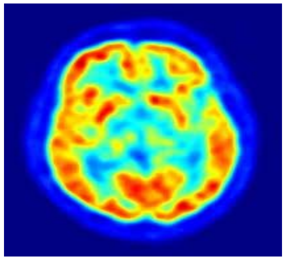
$C_i <= A_i * B_i$

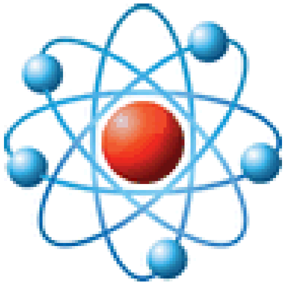
Ex: CRAY machine vector processing, Thinking machine cm*
Intel MMX (multimedia support),
Intel SSE/AVX (SIMD extensions)

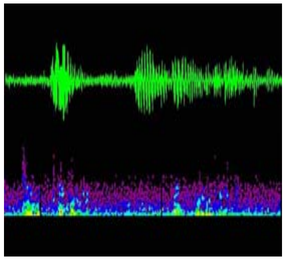
Adapted from Peh

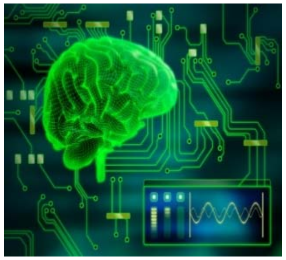
Data Parallelism is everywhere


Financial Modeling


Medical Imaging


Physics Simulation


Audio Processing


Machine Learning


Games


Image Processing


Statistics


Video Processing

- Mostly regular applications
- Works on large data sets

Adapted from Samadi, ASPLOS 2014

Outline

- GPU hardware introduction
- GPU programming introduction
- Programming challenges & current research

GPU: Highly Parallel Coprocessor

- GPU as a coprocessor that
  - Has its own DRAM memory
  - Communicate with host (CPU) through bus (PCIx)
  - Runs many threads in *parallel*
- GPU threads
  - GPU threads are extremely lightweight (almost no cost for creation/context switch)
  - GPU needs at least several thousands threads for full efficiency

Adapted from Utah SCI Institute

What is the GPU Good at?

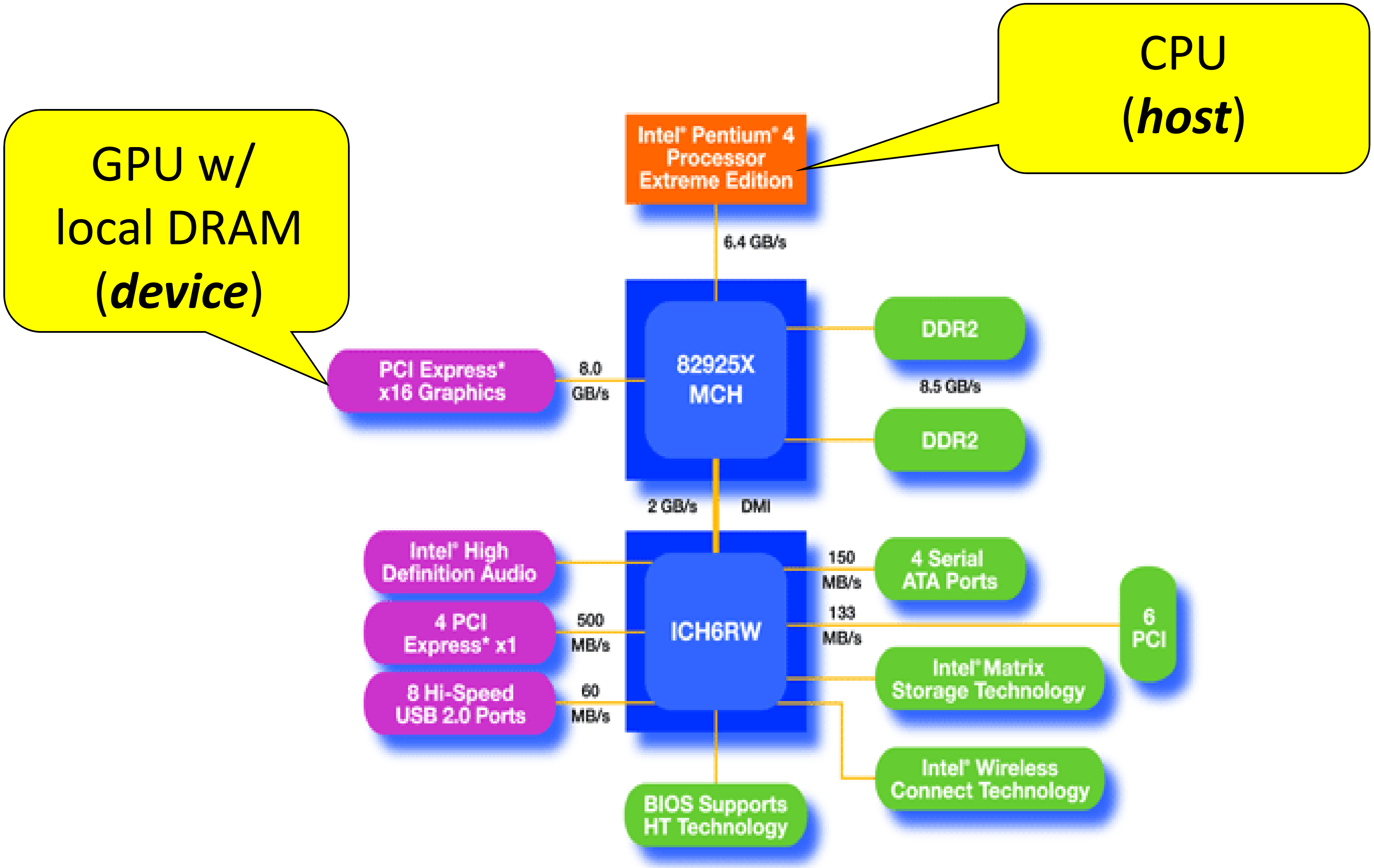
- The GPU is good at data-parallel processing
  - The same computation executed on many data elements in parallel – low control flow overhead

  with high SP floating point arithmetic intensity
  - Many calculations per memory access
  - Currently also need high floating point to integer ratio

- High floating-point arithmetic intensity and many data elements mean that **memory access latency can be hidden with calculations** instead of big data caches – Still need to avoid bandwidth saturation!

Adapted from Peh

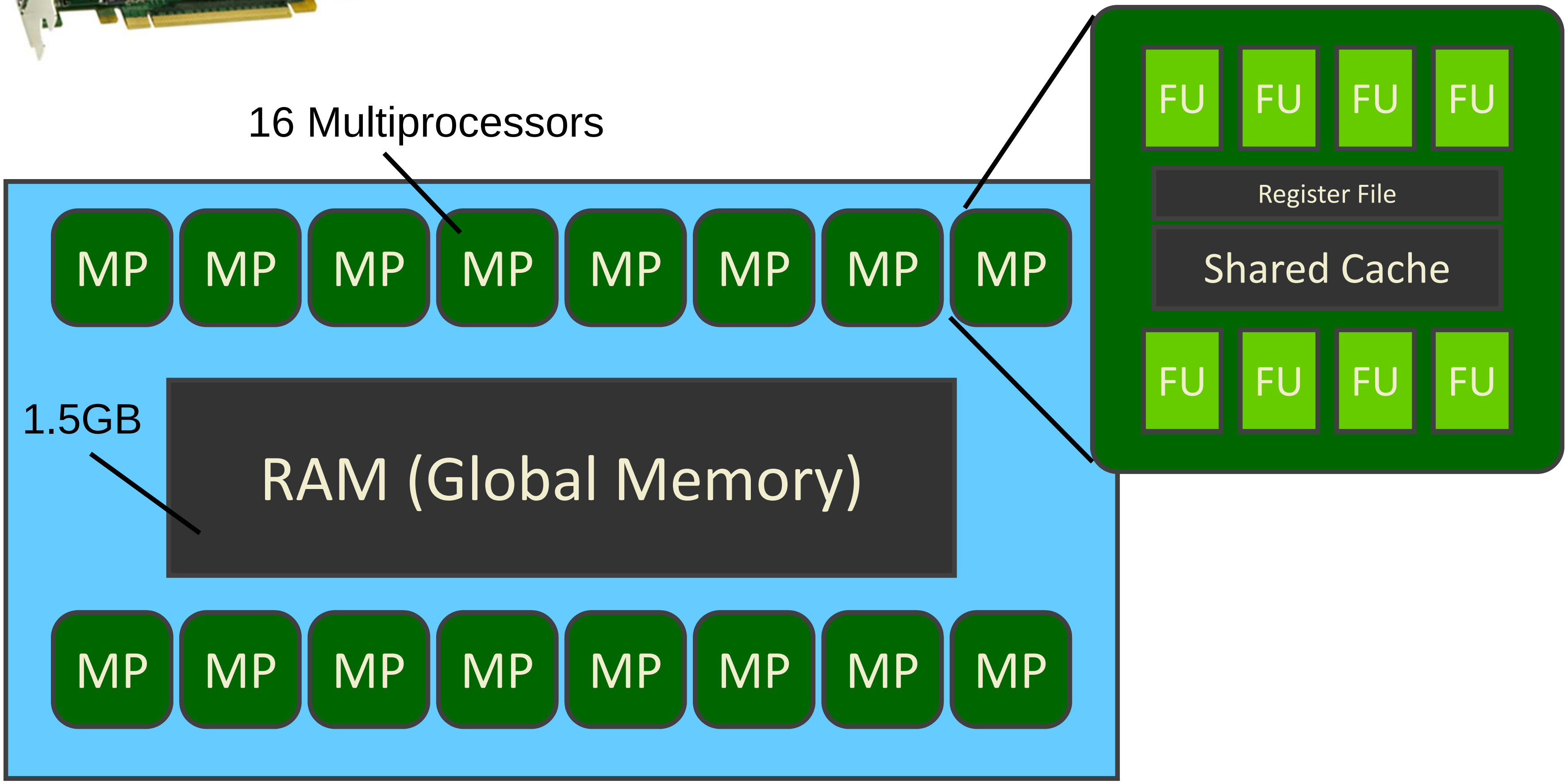
Example GPGPU System



GPU w/
local DRAM
(***device***)

CPU
(***host***)

Example GPU: Tesla



16 Multiprocessors

MP MP MP MP MP MP MP MP

1.5GB

RAM (Global Memory)

MP MP MP MP MP MP MP MP

FU FU FU FU

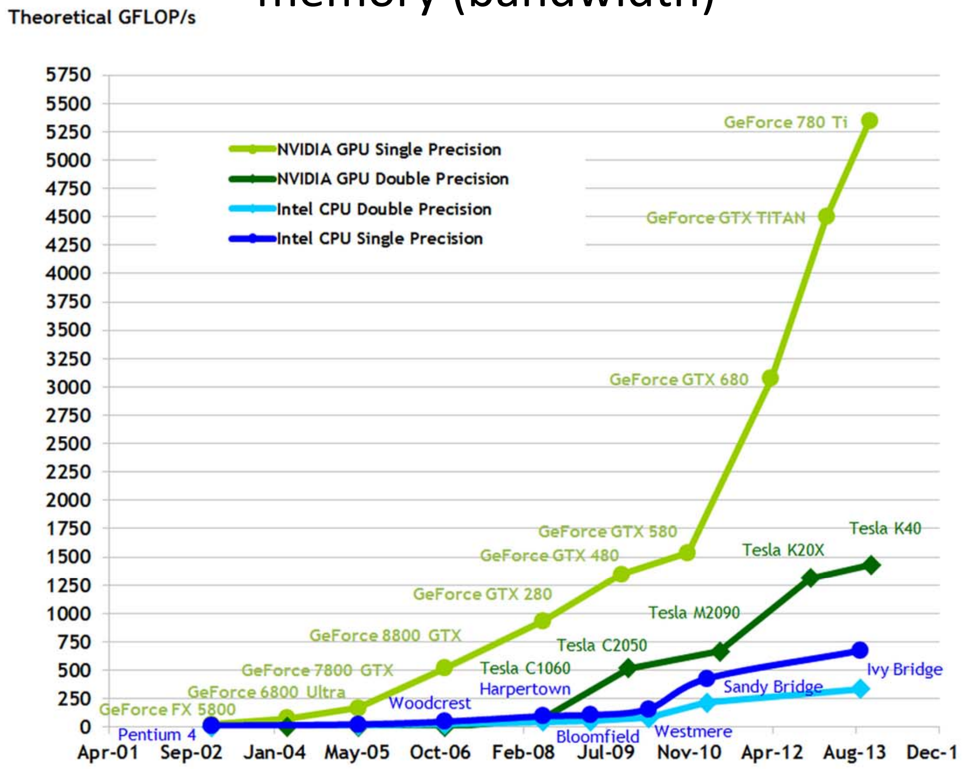
Register File

Shared Cache

FU FU FU FU

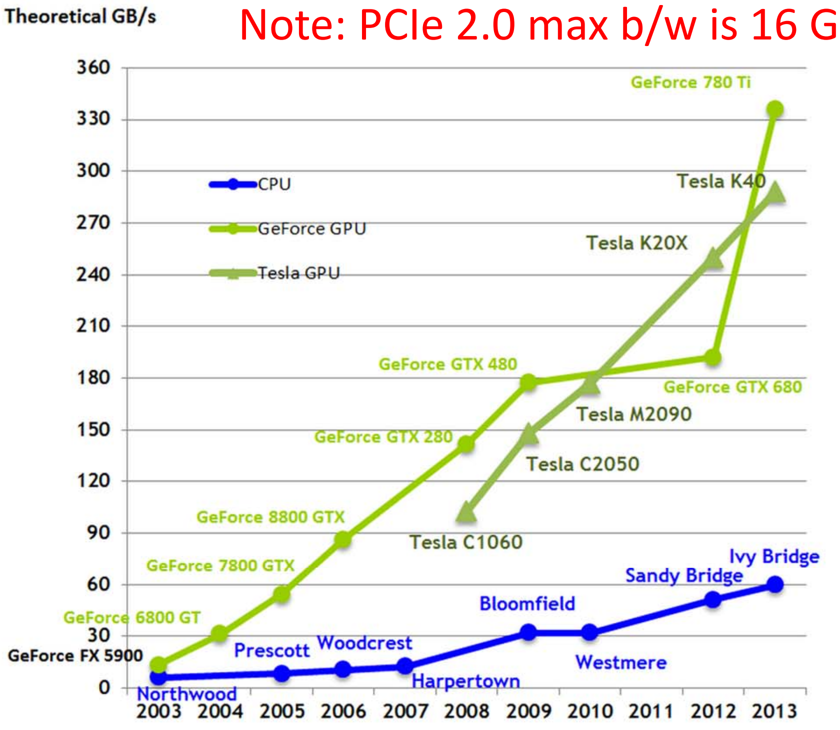
Why GPU for computing?

- Inexpensive supercomputers
- GPU hardware performance increases faster than CPU
  - Trend: simple, scalable architecture, interaction of clock speed, cache, memory (bandwidth)



Floating-Point Operations per Second - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Note: PCIe 2.0 max b/w is 16 GB/s

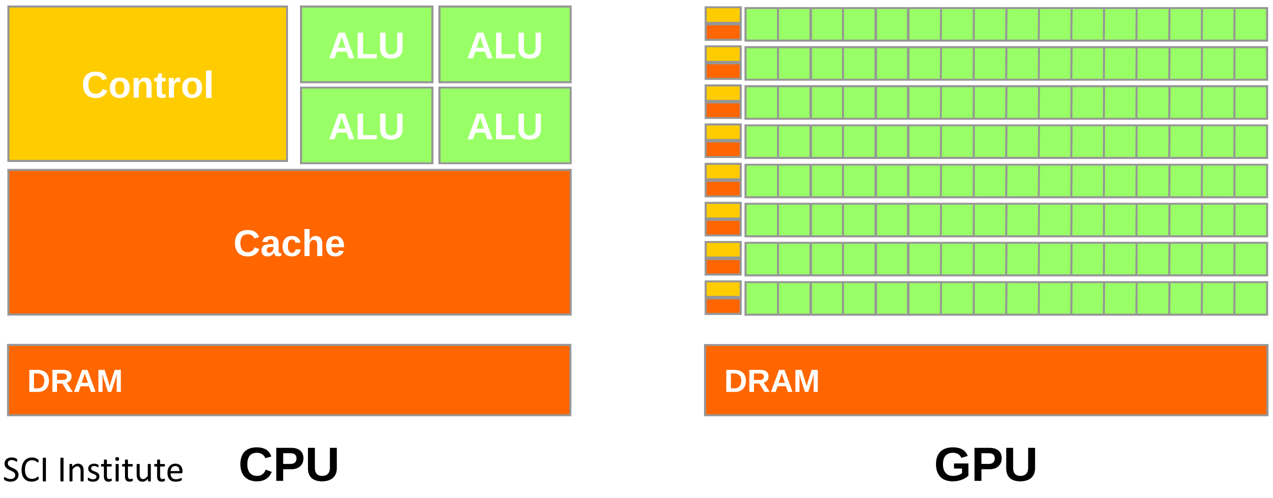


Memory Bandwidth for CPU and GPU - Nvidia CUDA C Programming Guide
Version 6.5 - 24/9/2014 - copyright Nvidia Corporation 2014

Adapted from Utah SCI Institute

GPU is for Parallel Computing

- CPU
  - Large cache and sophisticated flow control minimize latency for arbitrary memory access for serial process
- GPU
  - Simple flow control and limited cache, more transistors for computing in parallel
  - High arithmetic intensity hides memory latency

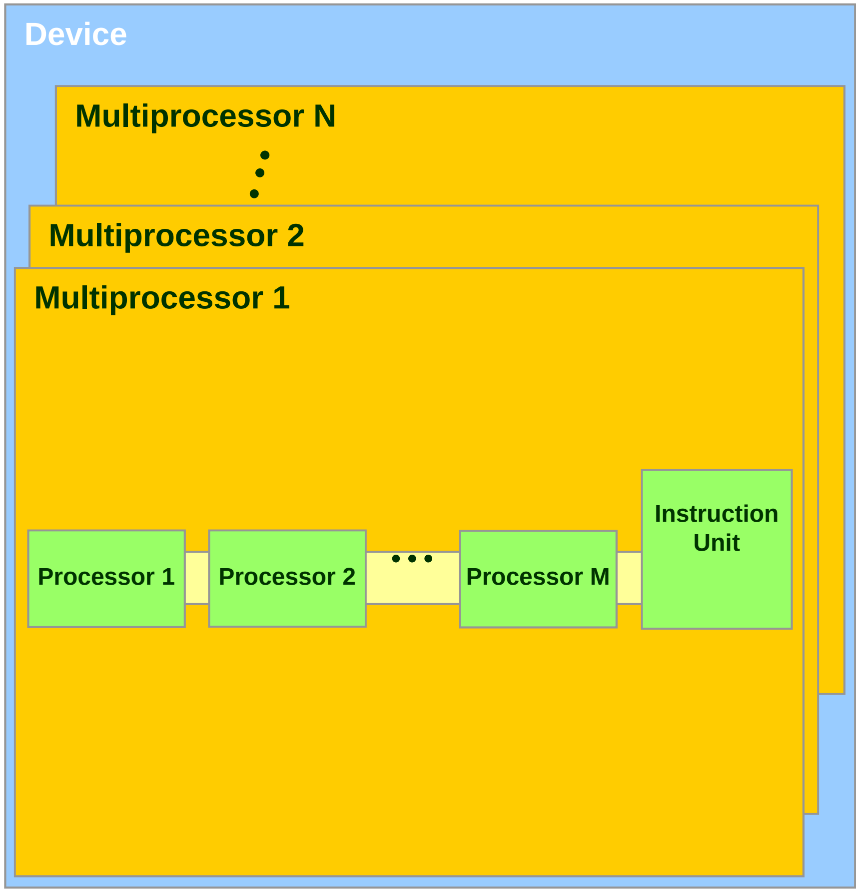


Courtesy NVIDIA

Adapted from Utah SCI Institute

Hardware Implementation :
a set of SIMD Processors

- Device
  - – a set of multiprocessors
- Multiprocessor
  - – a set of 32-bit SIMD processors

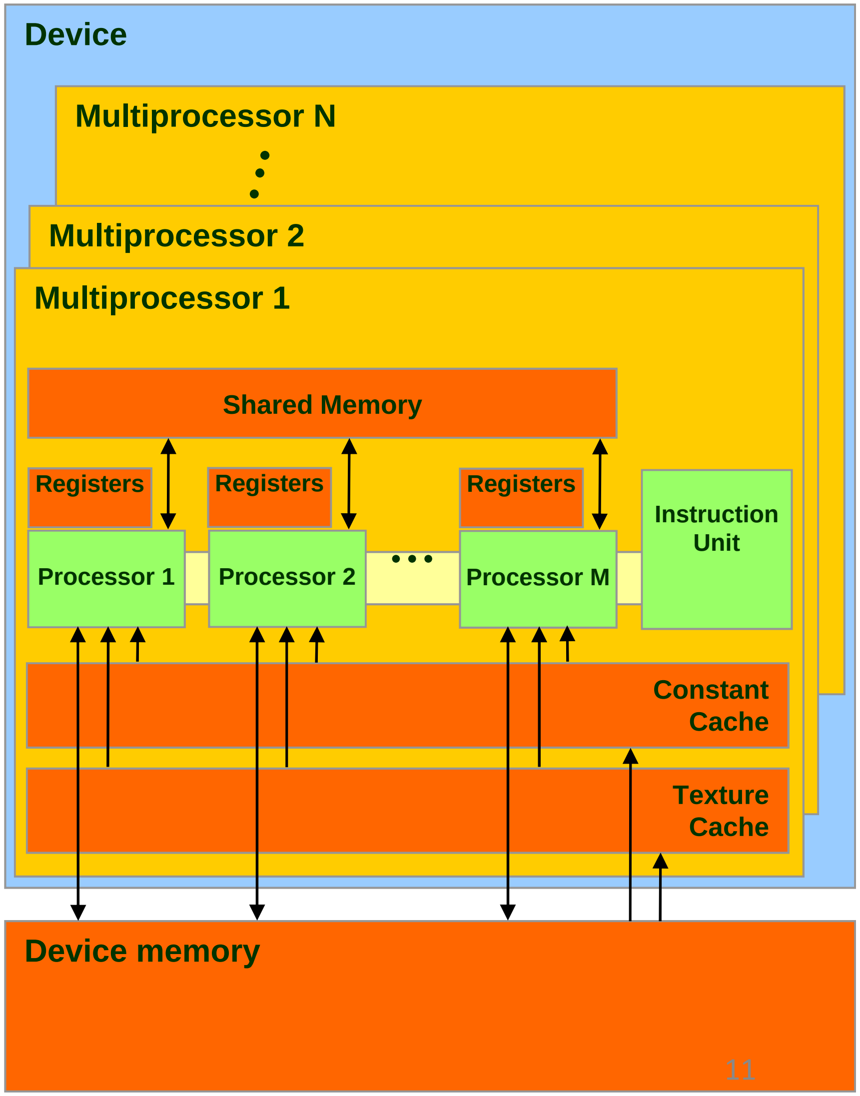


Courtesy NVIDIA

Adapted from Utah SCI Institute

Hardware Implementation: Memory Architecture

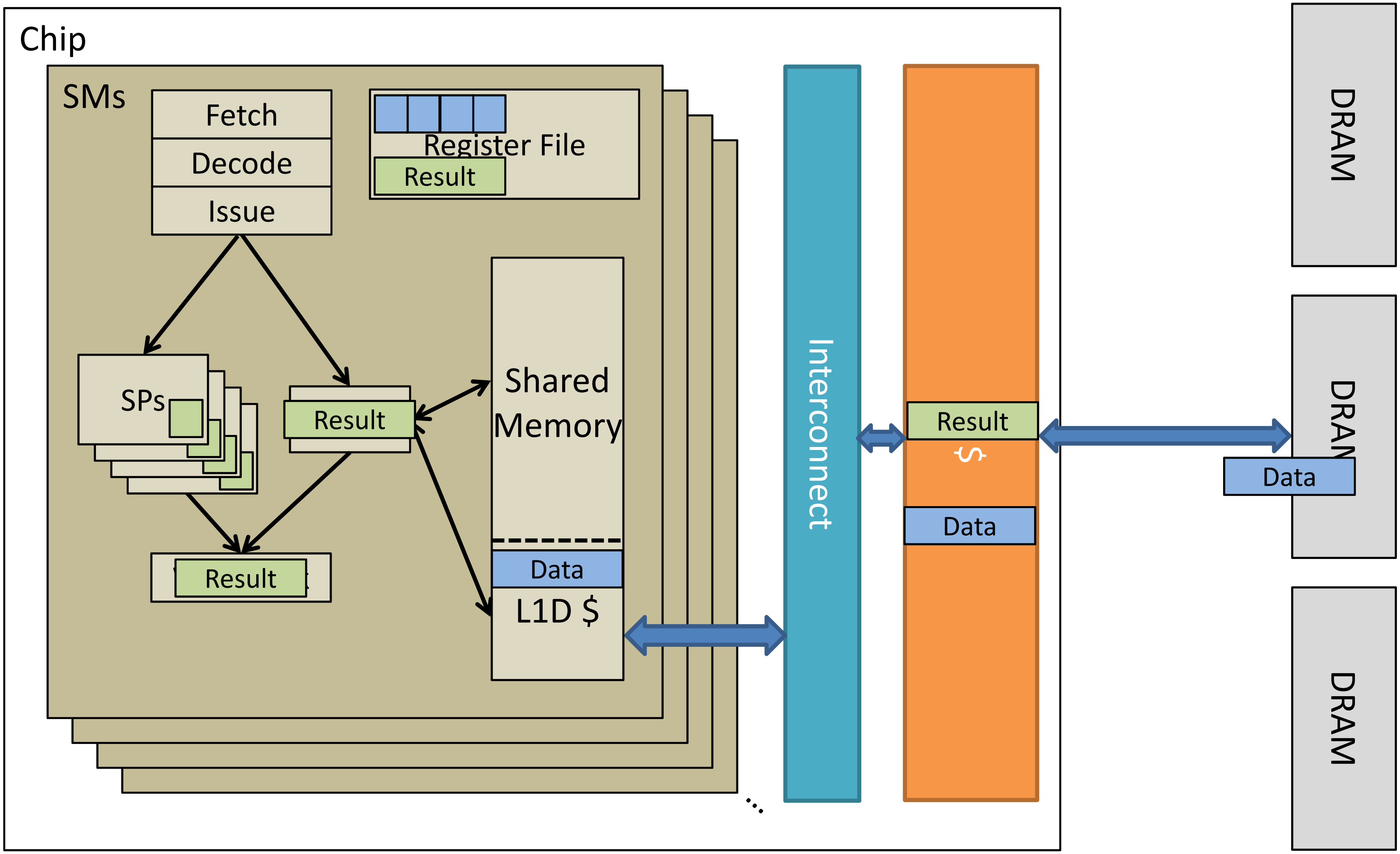
- Device memory (DRAM)
  - Slow (~300-400 cycles)
  - Local, global, constant, and texture memory
- On-chip memory
  - Fast (<10 cycles)
  - Registers, shared memory, constant/texture cache



Courtesy NVIDIA

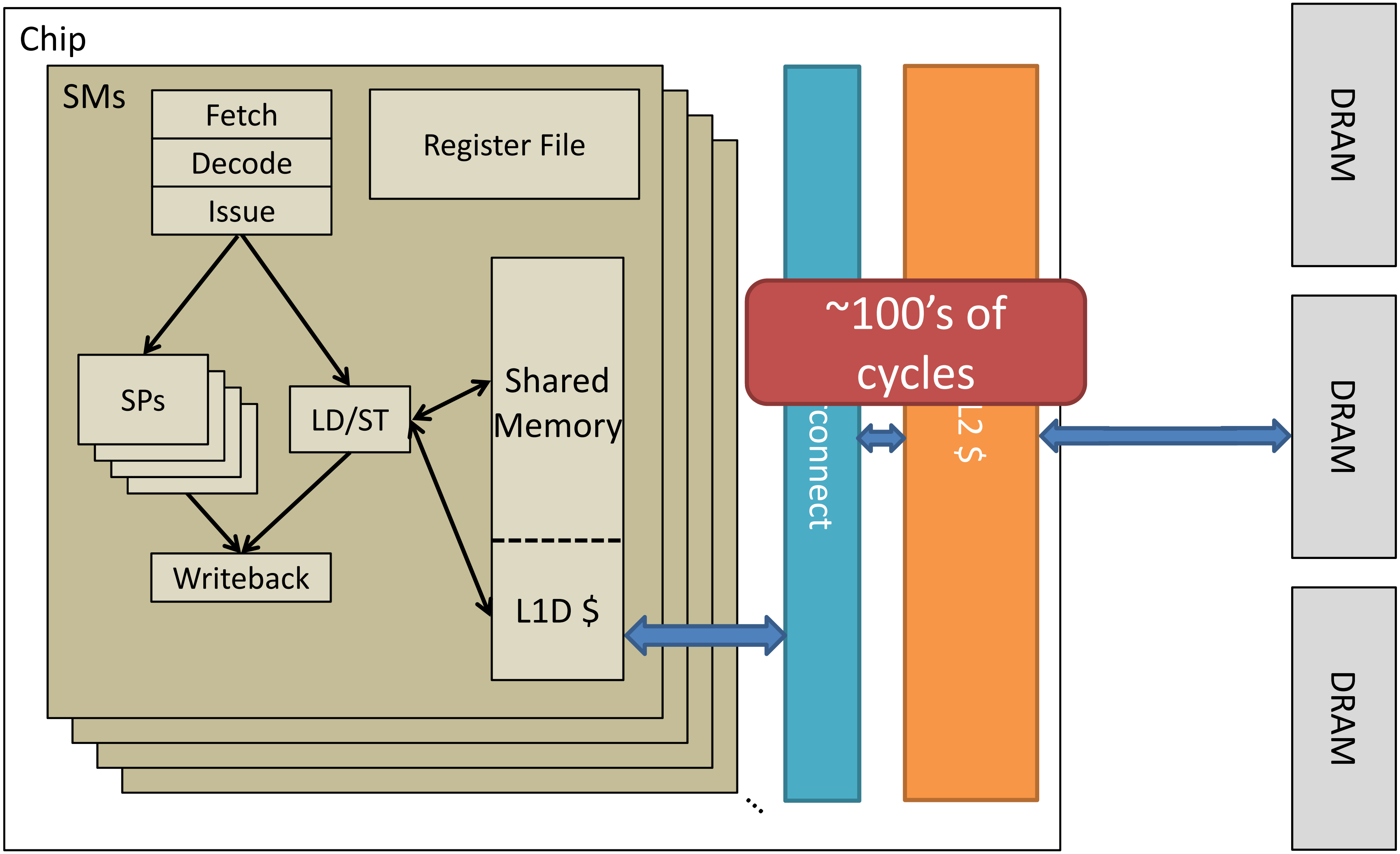
Adapted from Utah SCI Institute

A Quick Overview of GPUs



Adapted from Jamshidi, PACT 2014

A Quick Overview of GPUs

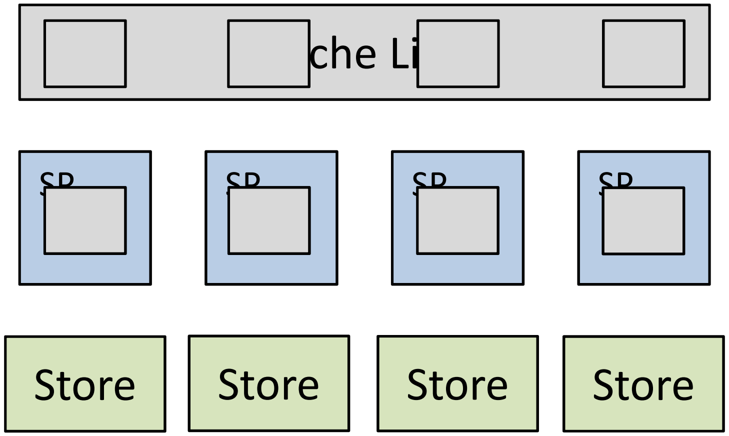


~100's of cycles

Adapted from Jamshidi, PACT 2014

How do GPUs Achieve Great Performance?

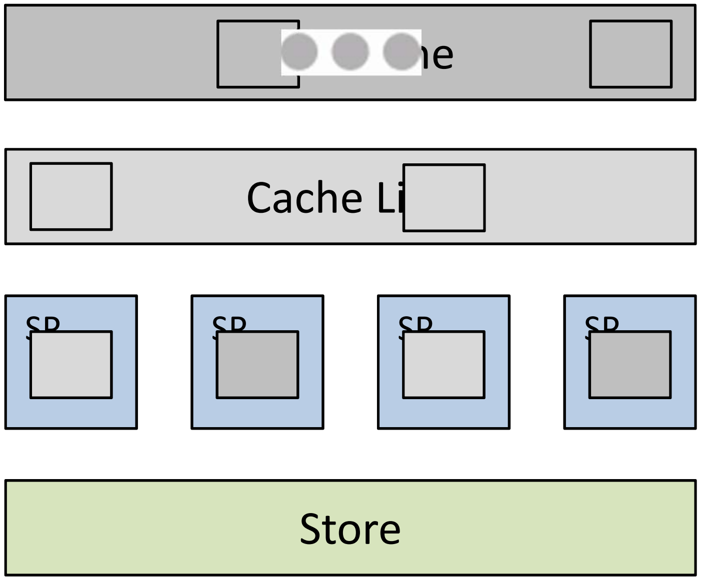
- Effectively use available memory bandwidth
  - Exploit data reuse when possible



Adapted from Jamshidi, PACT 2014

How do GPUs Achieve Great Performance?

- Effectively use available memory bandwidth
  - Exploit data reuse when possible
  - Regular, well coalesced memory accesses



Adapted from Jamshidi, PACT 2014

Outline

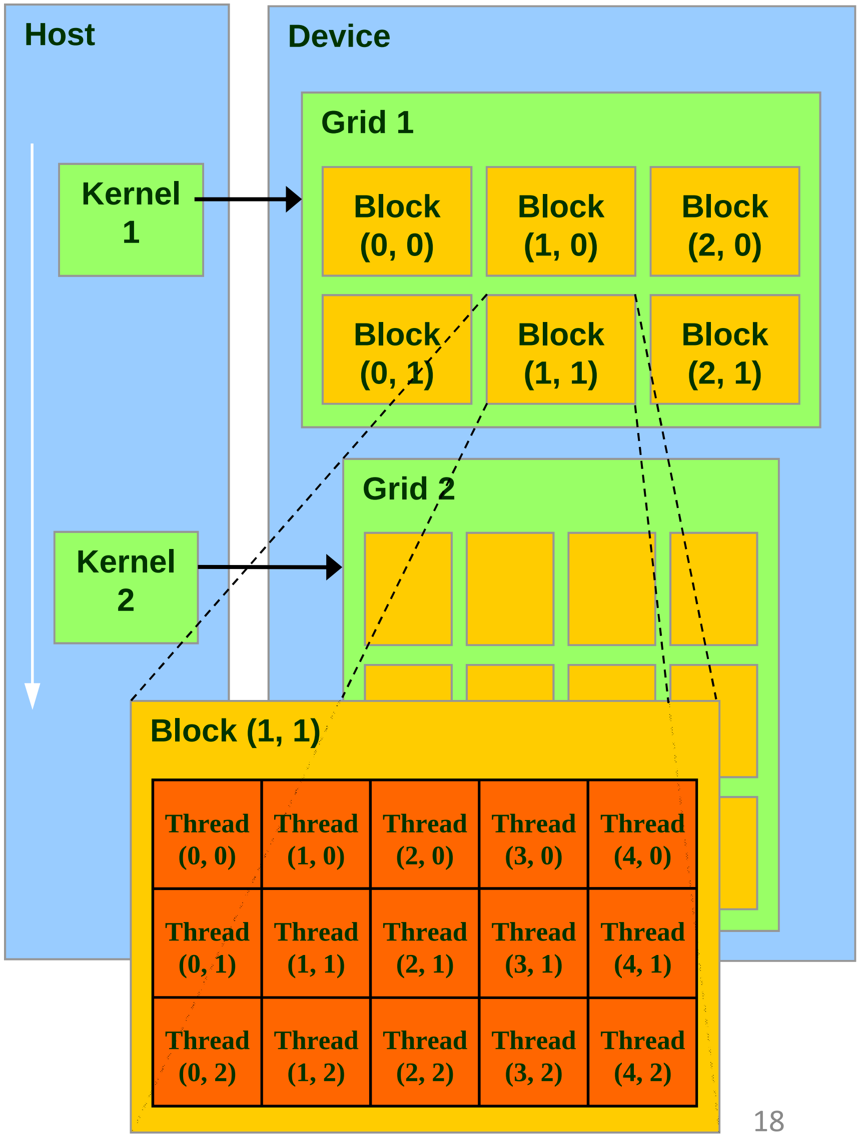
- GPU hardware introduction
- GPU programming introduction
- Programming challenges & current research

CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot directly cooperate



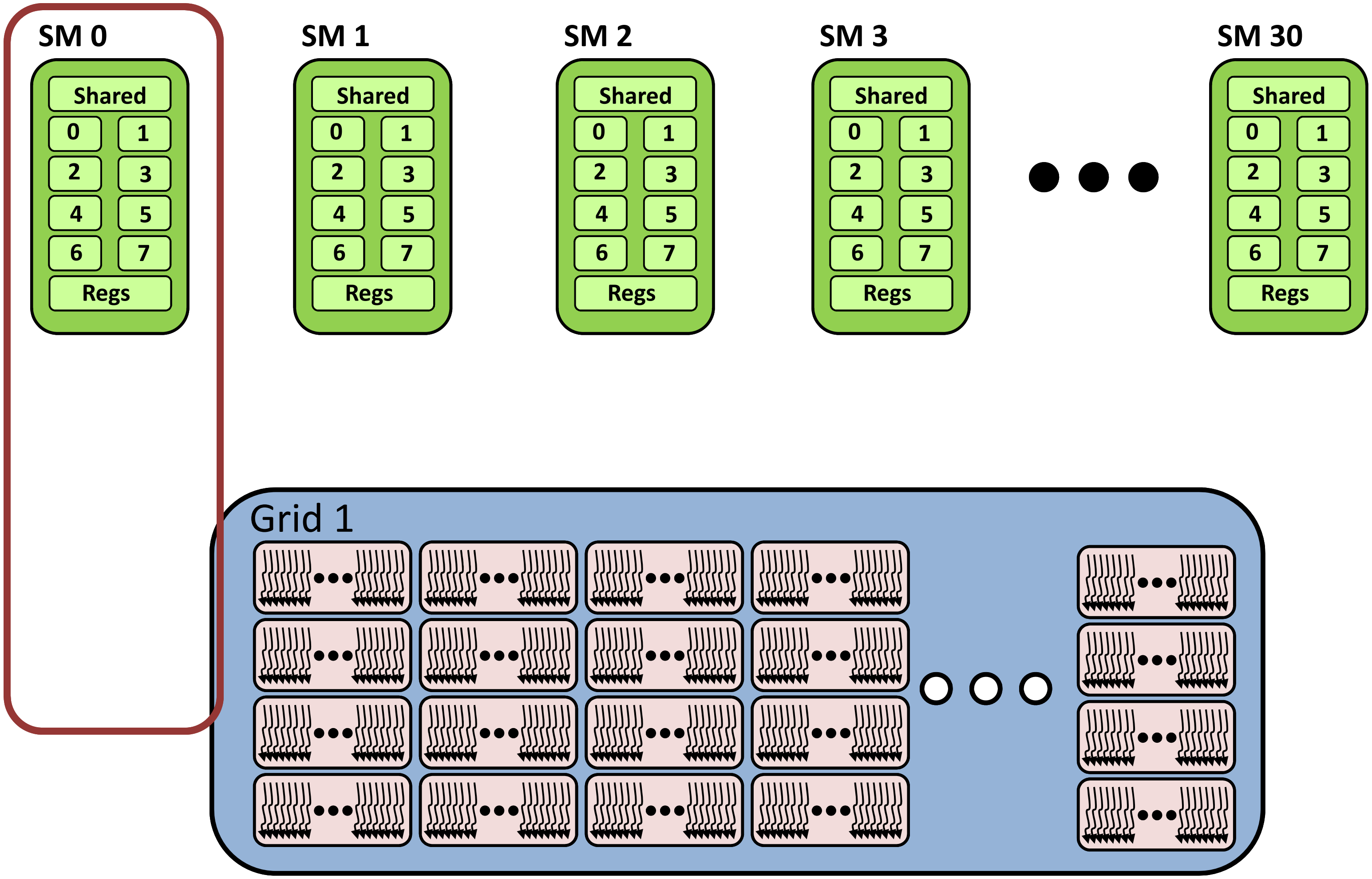
Courtesy: NVIDIA

Execution Model

- Each thread block is executed by a single multiprocessor
  - Synchronized using shared memory
- Many thread blocks are assigned to a single multiprocessor
  - Executed concurrently in a time-sharing fashion
  - Keep GPU as busy as possible
- Running many threads in parallel can hide DRAM memory latency
  - Global memory access : ~300-400 cycles

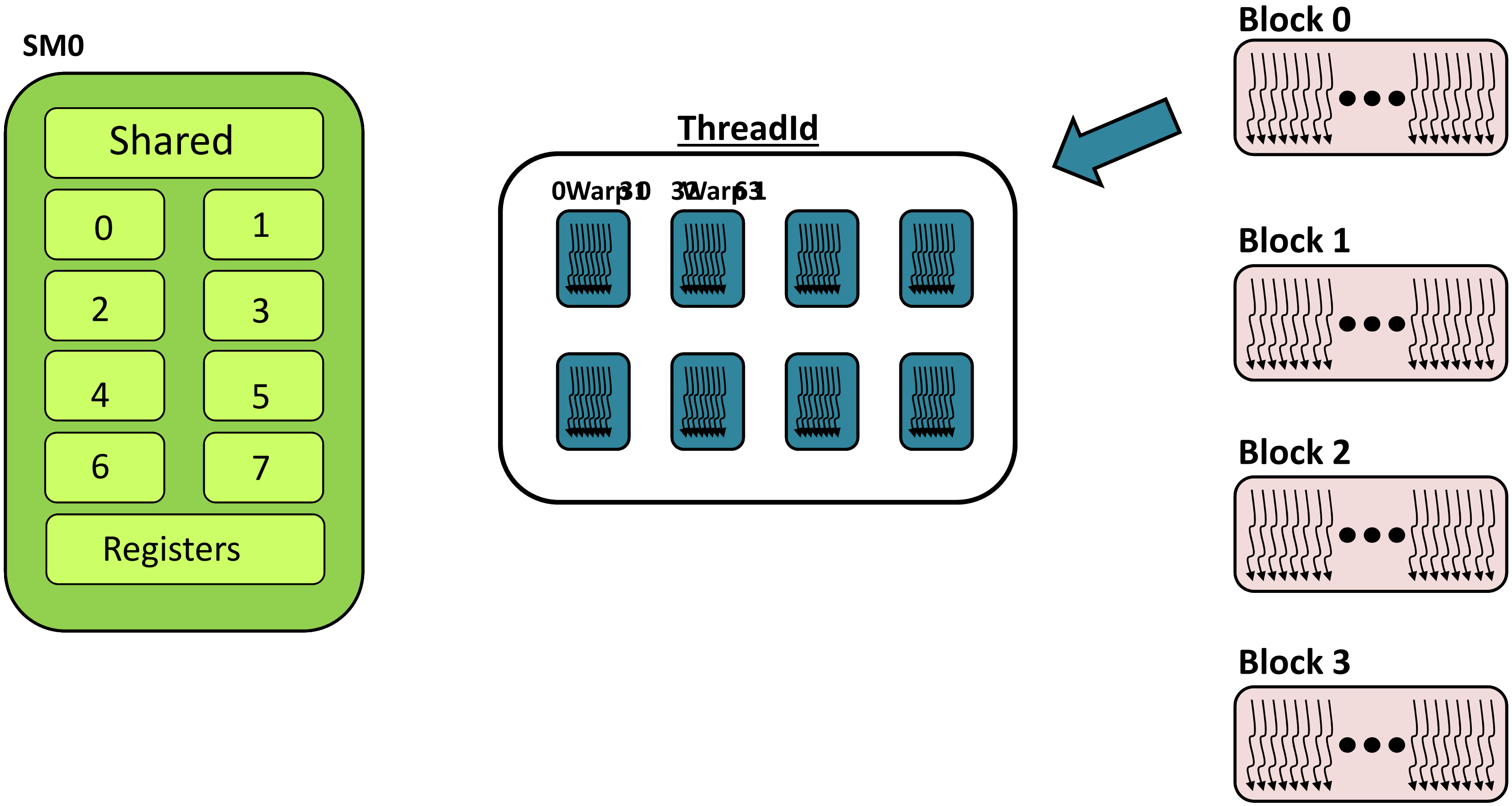
Adapted from Utah SCI Institute

GPU Execution Model



Adapted from Hormati, ASPLOS 2011

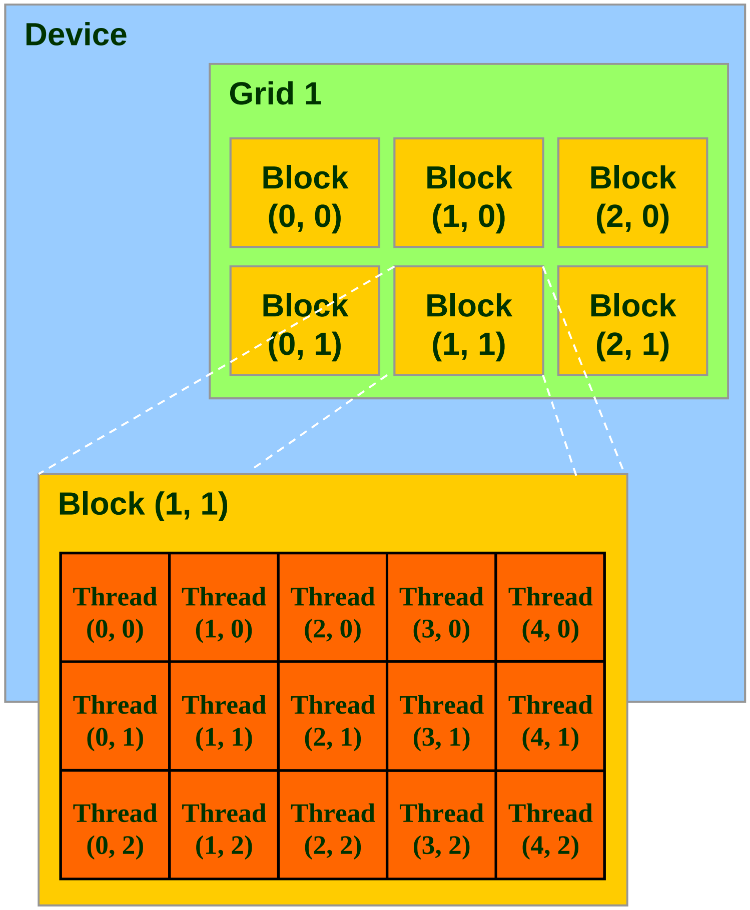
GPU Execution Model



Adapted from Hormati, ASPLOS 2011

Block and Thread IDs

- Threads and blocks have IDs
  - So each thread can decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …

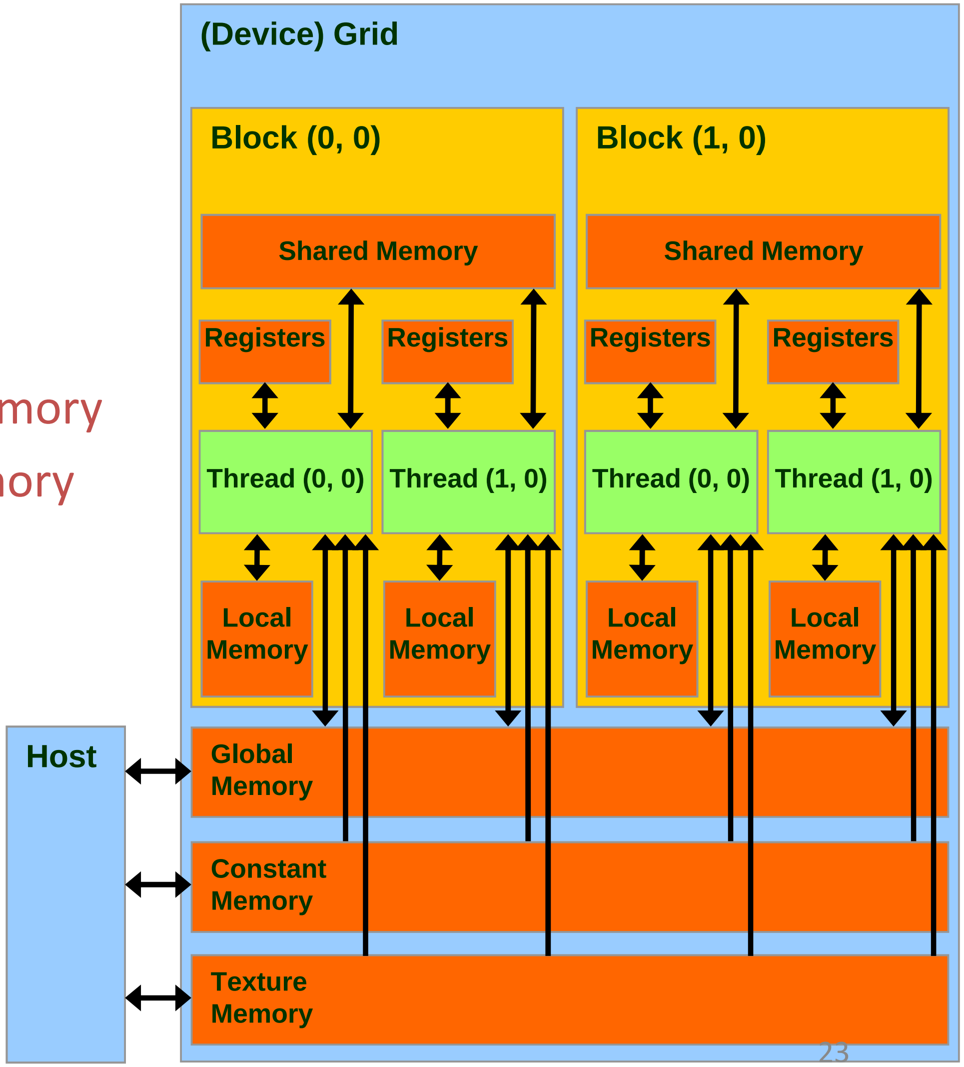


Courtesy: NDVIA

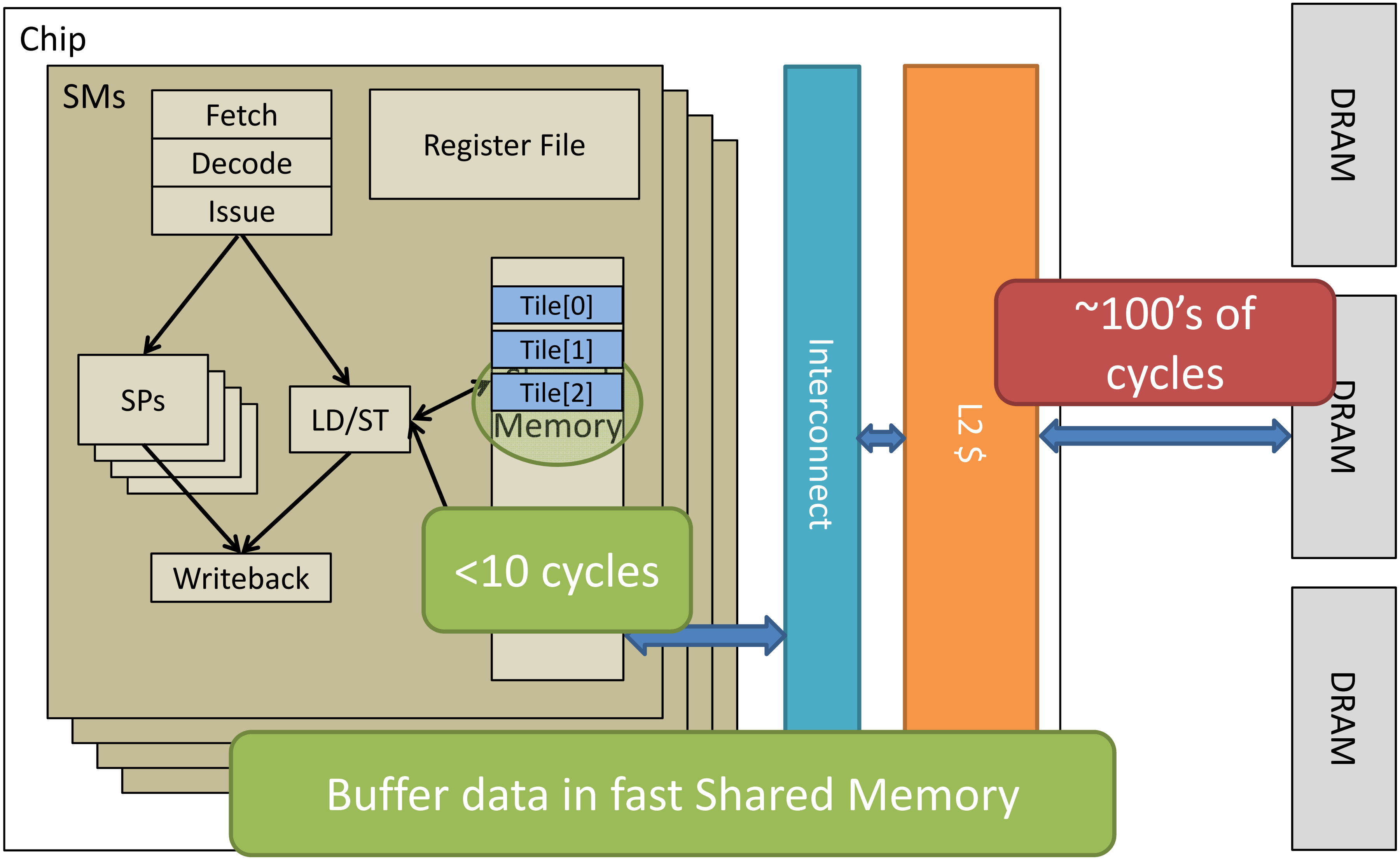
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories



Buffering to Optimize Bandwidth



~100's of cycles

<10 cycles

Buffer data in fast Shared Memory

Adapted from Jamshidi, PACT 2014

GeForce GTX 480 Technical Specs

- Maximum number of threads per block: 1024
- Maximum size of each dimension of a grid: 65,535
- Num … SM):
  - 1
- Device memory: 1536 MB

Many parameters change across generations!

- Shared memory per multiprocessor: 16/48KB divided in 32 banks
- Transistors/Size: 3 Billion transistors/529 mm^2

Adapted from Peh

CUDA

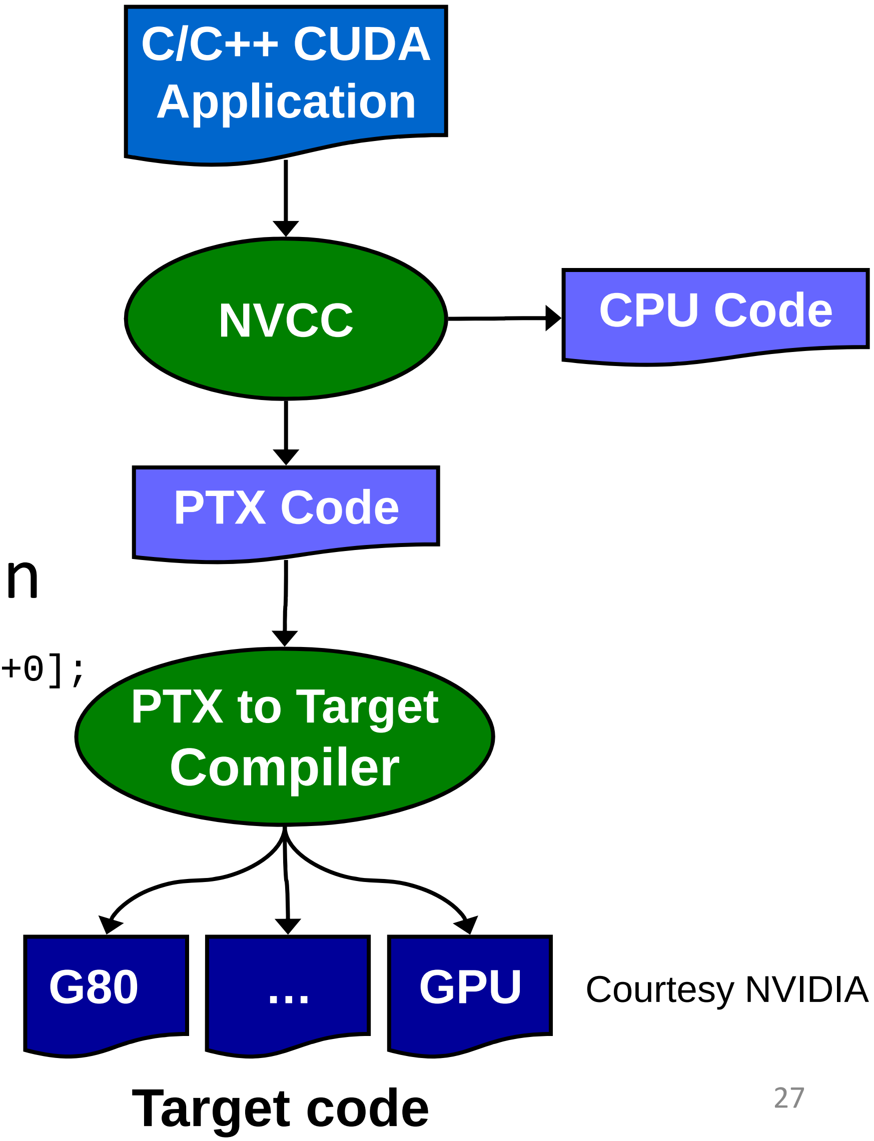
- C-extension programming language
  - No graphics API
    - Flattens learning curve
    - Better performance
  - Support debugging tools
- Extensions / API
  - Function type : __global__, __device__, __host__
  - Variable type : __shared__, __constant__
  - cudaMalloc(), cudaFree(), cudaMemcpy(),…
  - __syncthread(), atomicAdd(),…
- Program types
  - *Device* program (kernel) : run on the GPU
  - *Host* program : run on the CPU to call device programs

Adapted from Utah SCI Institute

Compiling CUDA

- nvcc
  - Compiler driver
  - Invoke cudacc, g++, cl
- PTX
  - Parallel Thread eXecution

```
ld.global.v4.f32  {$f1,$f3,$f5,$f7}, [$r9+0];
mad.f32           $f1, $f5, $f3, $f1;
```



Courtesy NVIDIA

Adapted from Utah SCI Institute

Extended C

- **Declspecs**
  - **global, device, shared, local, constant**
- **Keywords**
  - **threadIdx, blockIdx**
- **Intrinsics**
  - **__syncthreads**
- **Runtime API**
  - **Memory, symbol, execution management**
- **Function launch**

```
__device__ float filter[N];

__global__ void convolve (float *image)  {

  __shared__ float region[M];
  ...

  region[threadIdx] = image[i];

  __syncthreads()
  ...

  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)


// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```

CUDA Function Declarations

	Executed on the:	Only callable from the:
`__device__ float DeviceFunc()`	device	device
`__global__ void KernelFunc()`	device	host
`__host__ float HostFunc()`	host	host

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together
- `__device__` functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Language Extensions: Built-in Variables

- `dim3 gridDim;`
  - Dimensions of the grid in blocks (`gridDim.z` unused)
- `dim3 blockDim;`
  - Dimensions of the block in threads
- `dim3 blockIdx;`
  - Block index within the grid
- `dim3 threadIdx;`
  - Thread index within the block

Adapted from Peh

Example: Vector Addition Kernel

```
// Pair-wise addition of vector elements
// One thread per addition

__global__ void
vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x
        + blockDim.x * blockId.x;
    oC[idx] = iA[idx] + iB[idx];
}
```

Courtesy NVIDIA

Adapted from Utah SCI Institute

Example: Vector Addition Host Code

```
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// … initalize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float),
   cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, N * sizeof(float),
   cudaMemcpyHostToDevice );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```

Courtesy NVIDIA

Adapted from Utah SCI Institute

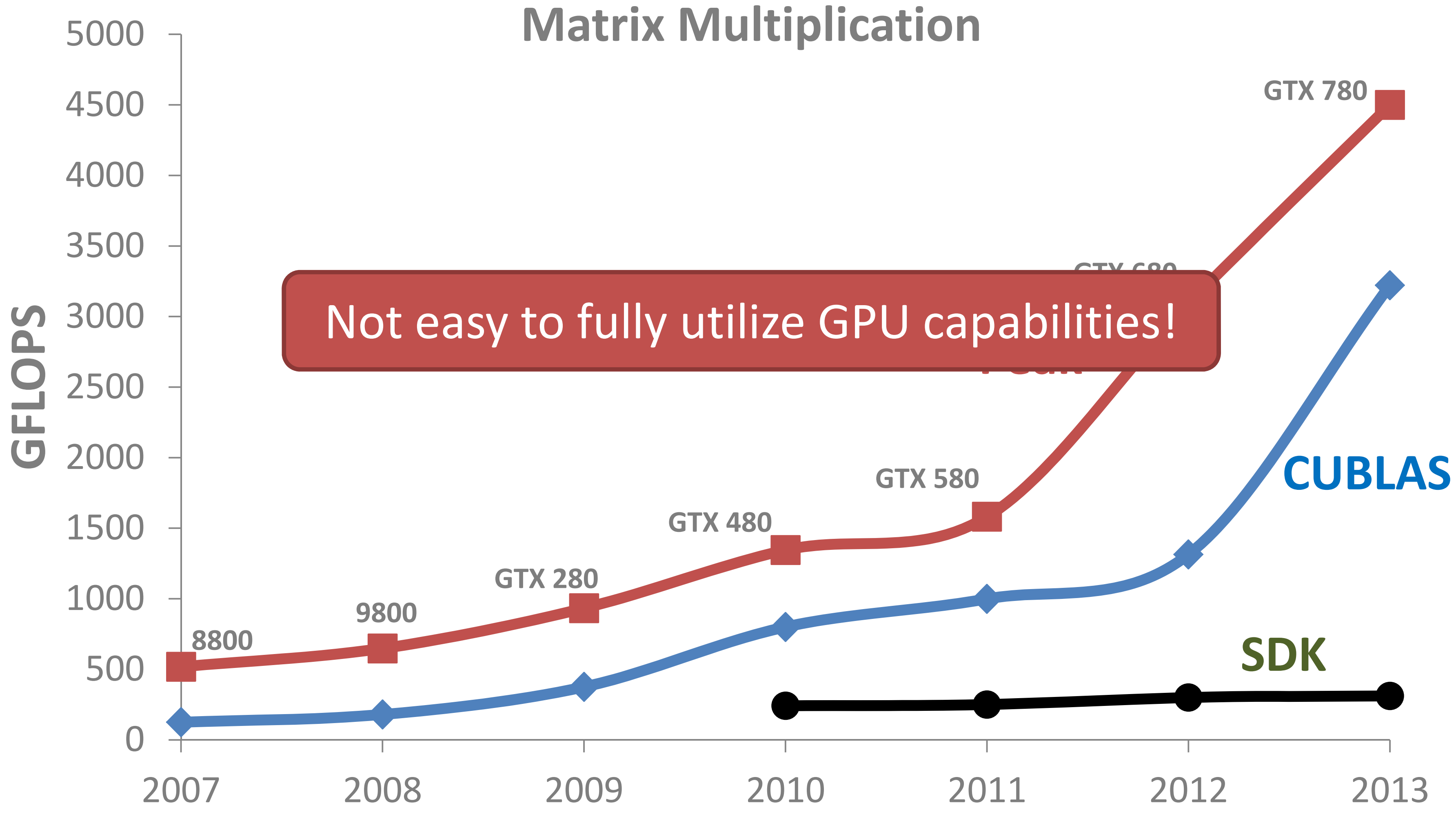
For more CUDA info…

- Vector types
- Atomic operations
- Thread synchronization
- Texture cache usage
- Mathematical functions
- API details
- … see the **NVIDIA CUDA C Programming Guide**
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Outline

- GPU hardware introduction
- GPU programming introduction
- Programming challenges & current research

Achieving Peak GPU Performance: Theory and Practice

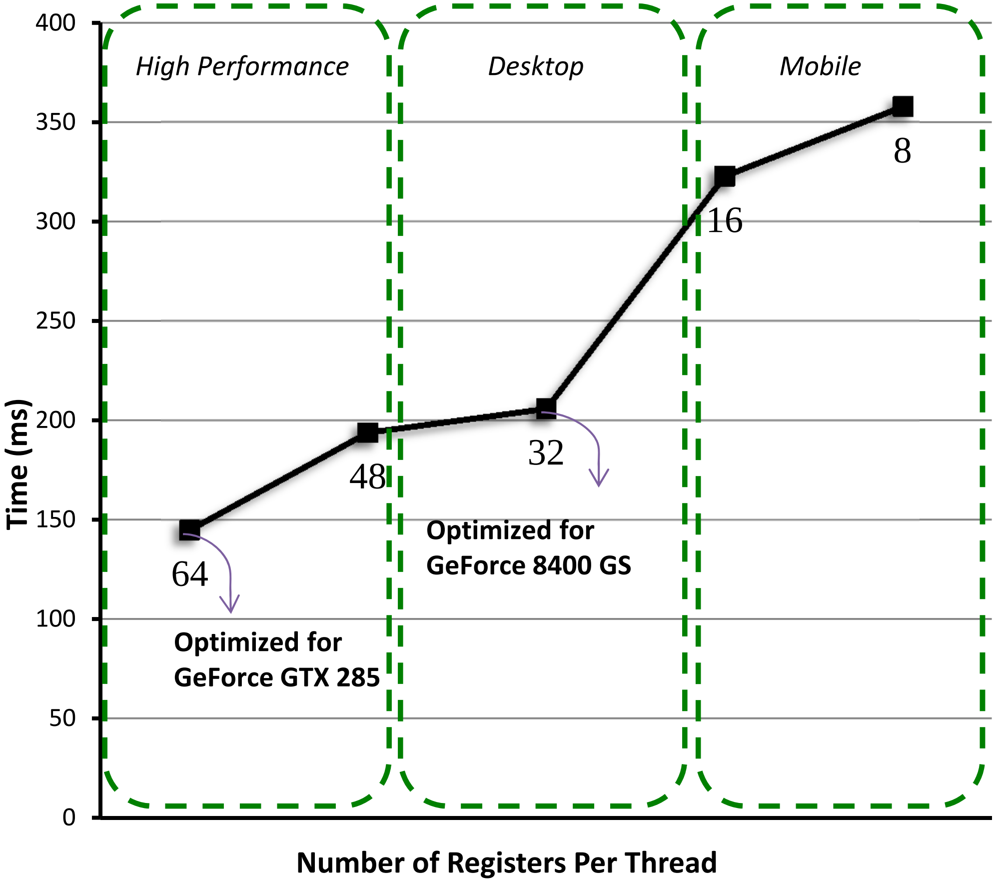


Not easy to fully utilize GPU capabilities!

Adapted from Jamshidi, PACT 2014

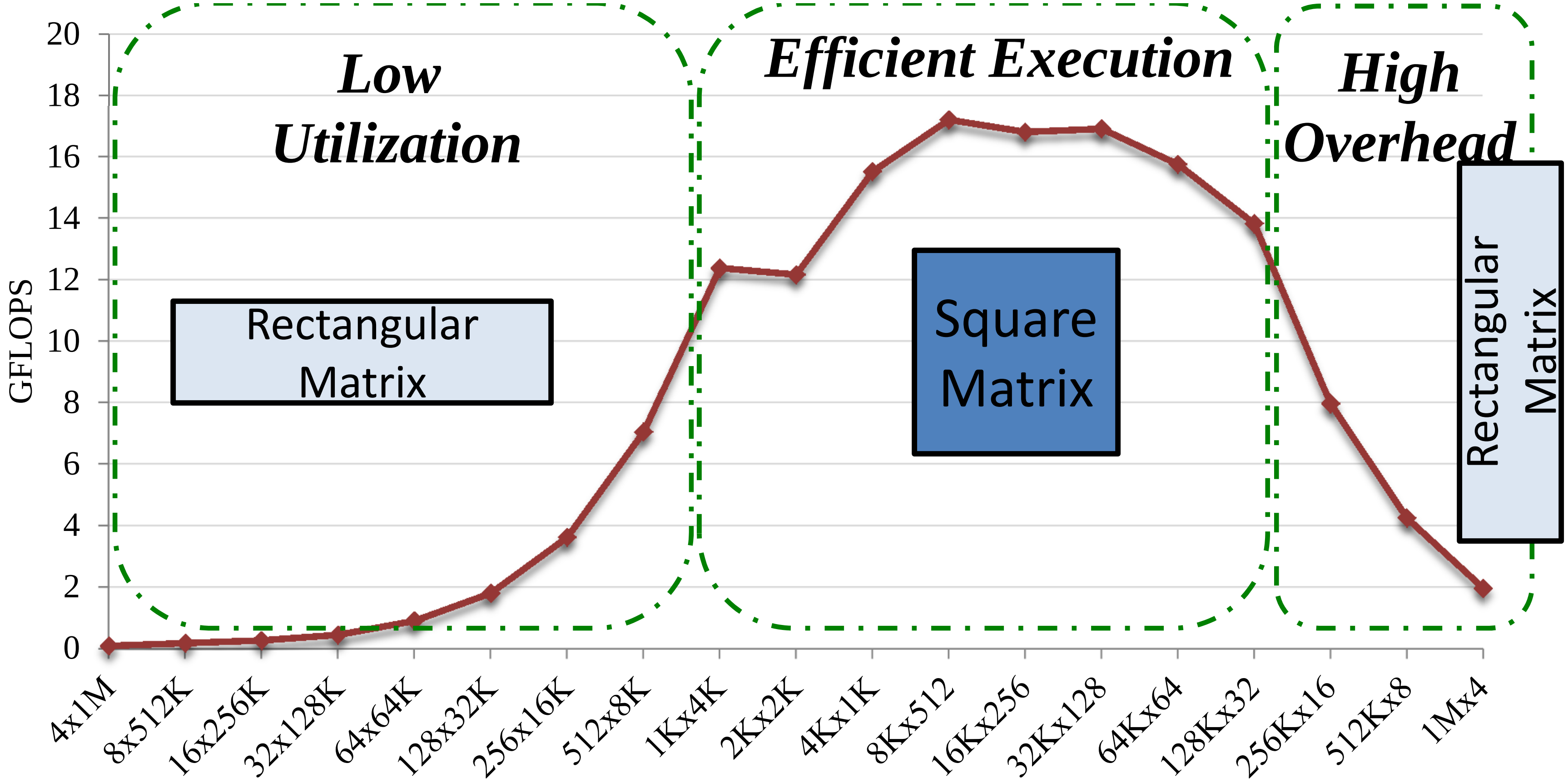
GPU Programming Challenges

- Data restructuring for complex memory hierarchy efficiently
  - Global memory, Shared memory, Registers
- Partitioning work between CPU and GPU
- Lack of portability between different generations of GPU
  - Registers, active warps, size of global memory, size of shared memory
- Will vary even more
  - Newer high performance cards e.g. NVIDIA's Kepler, Maxwell…
  - Mobile GPUs with fewer resources



Adapted from Hormati, ASPLOS 2011

Varying Inputs, Drastic Performance Effects

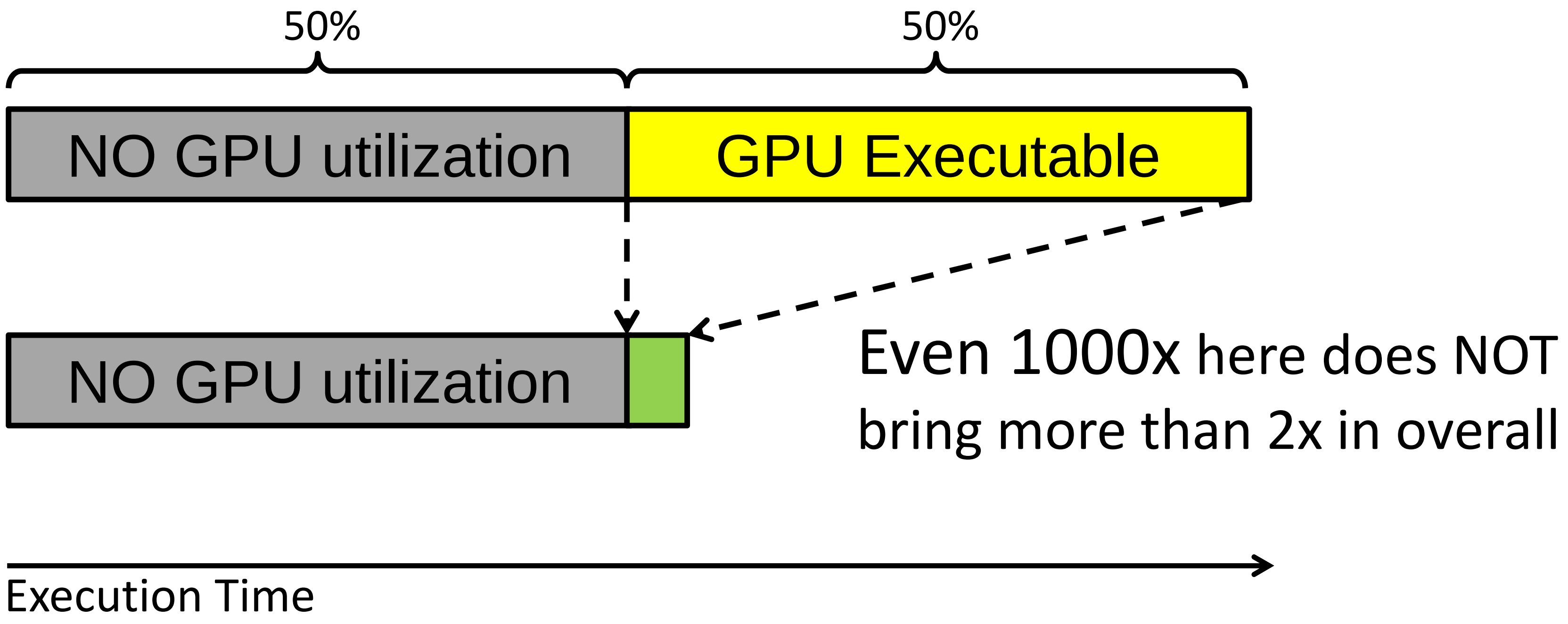


Optimize by writing many kernels?
Compiler support may ease this burden.

Adapted from Samadi, PLDI 2012

Amdahl’s Law

- GPGPU may have <100x speedup but...



Even 1000x here does NOT bring more than 2x in overall

Adapted from Samadi, GPGPU 2012

General Purpose Computing on GPU

- Limitation of GPU Executable
  - Massive Data-Parallelism
  - Line
  - NO
  - NO Pointers

How can GPUs be more GENERAL?

- Leaves GPUs underutilized
  - GPGPUs are not general enough

Adapted from Samadi, GPGPU 2012

Barriers to Generalization

- Reduce **NO GPU utilization** Sections
  - Non-Linear array access

```
for(y=0; y<ny; y++)
```

```
for(i=1; i<m; i++)
```

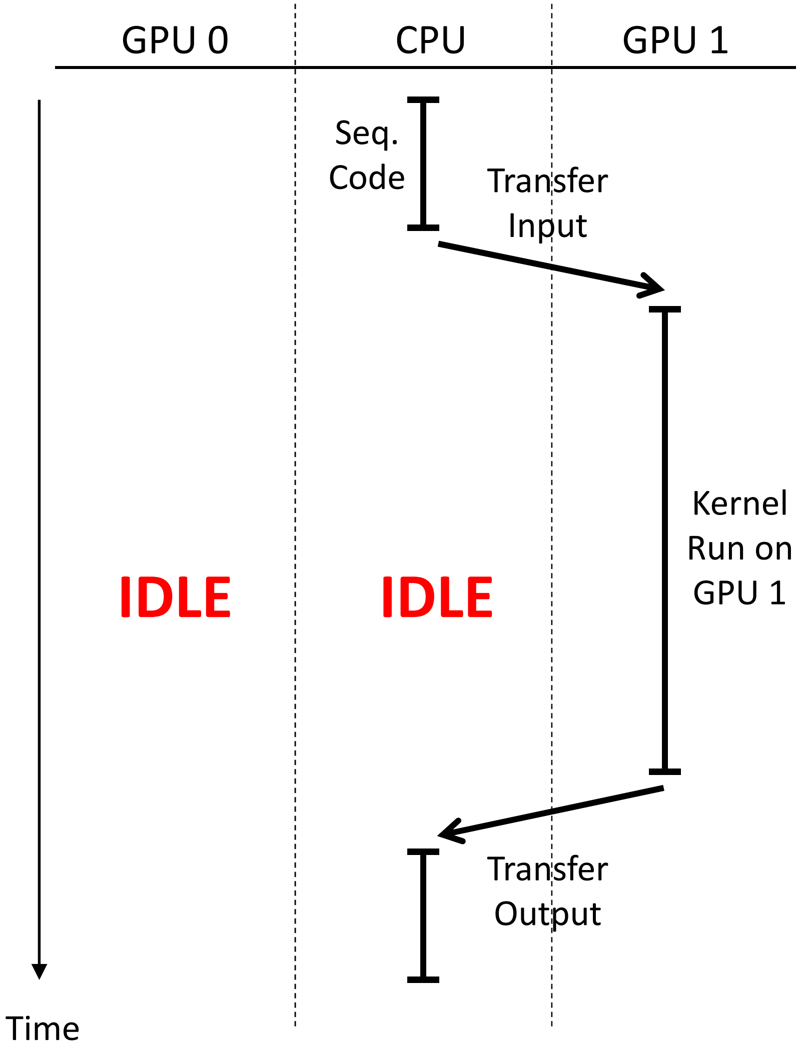
```
for(int i=0; i<n; i++){
   *c = *a + *b;
   a++;  b++;  c++;
}
```

```
   }
```

cial…

Adapted from Samadi, GPGPU 2012

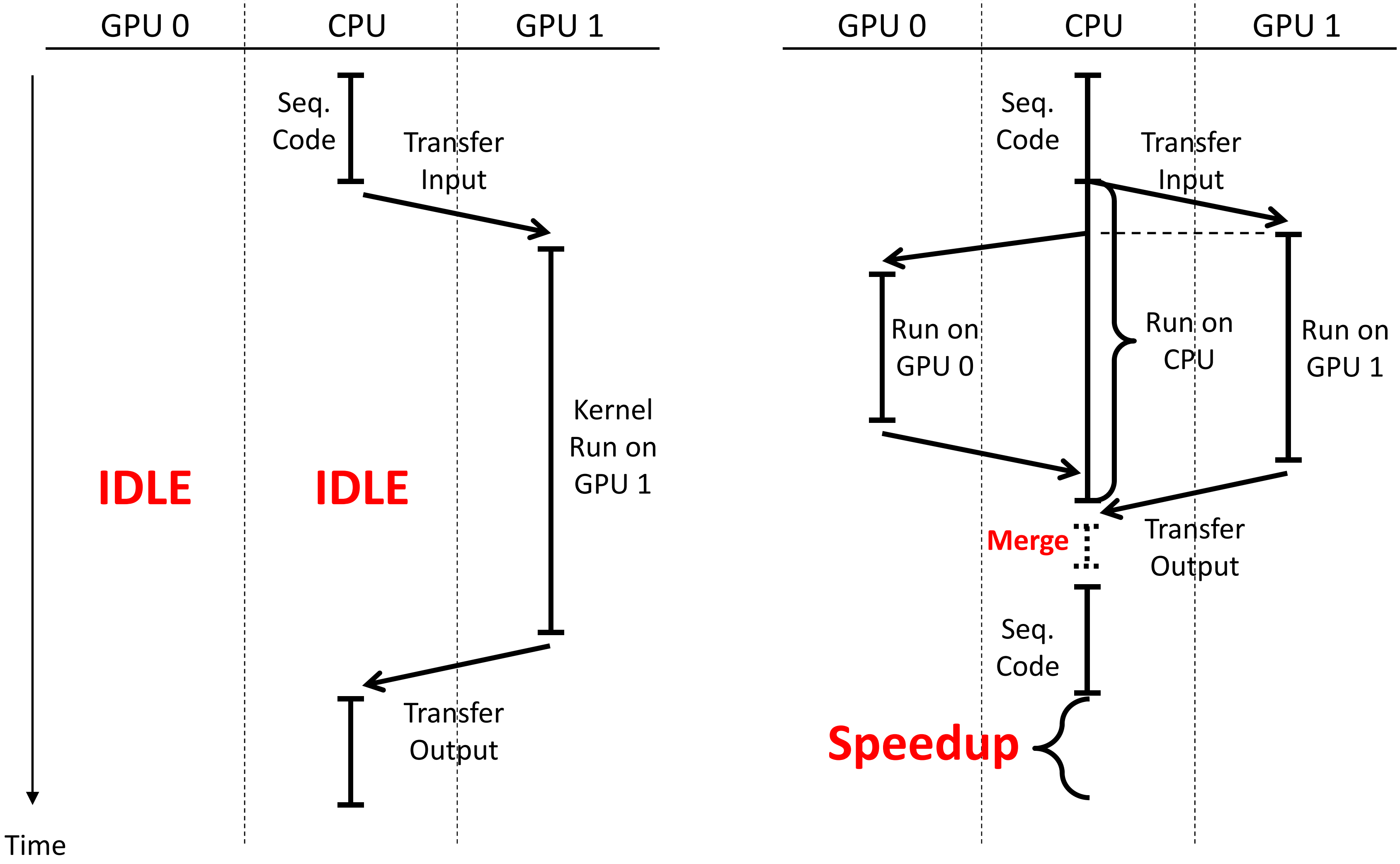
What about Heterogeneity?



Why not utilize GPU 0 and CPU's SIMD resources?

Adapted from Lee, PACT 2013

Collaborative Heterogeneous Execution



GPU 0

CPU

GPU 1

Seq. Code

Transfer Input

Kernel Run on GPU 1

IDLE

IDLE

Transfer Output

Time

GPU 0

CPU

GPU 1

Seq. Code

Transfer Input

Run on GPU 0

Run on CPU

Run on GPU 1

Merge

Transfer Output

Seq. Code

Speedup

Adapted from Lee, PACT 2013

Current Hardware Research

- Remember, trend is ***simple & scalable***
  - General goals:
    - Reducing cache thrashing
    - Better overlapping of computation & memory
    - Improved performance & energy efficiency
  - Warp scheduling
    - Cache-Conscious Wavefront Scheduling (Rogers et. al.)
  - Prefetching
    - Adaptive prefetching (APOGEE, Sethia et al.)