Scaling in a Heterogeneous Environment with GPUs: GPU Architecture, Concepts, and Strategies

John E. Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

http://www.ks.uiuc.edu/~johns/

Petascale Computing Institute,

National Center for Supercomputing Applications,

University of Illinois at Urbana-Champaign

GPU Computing

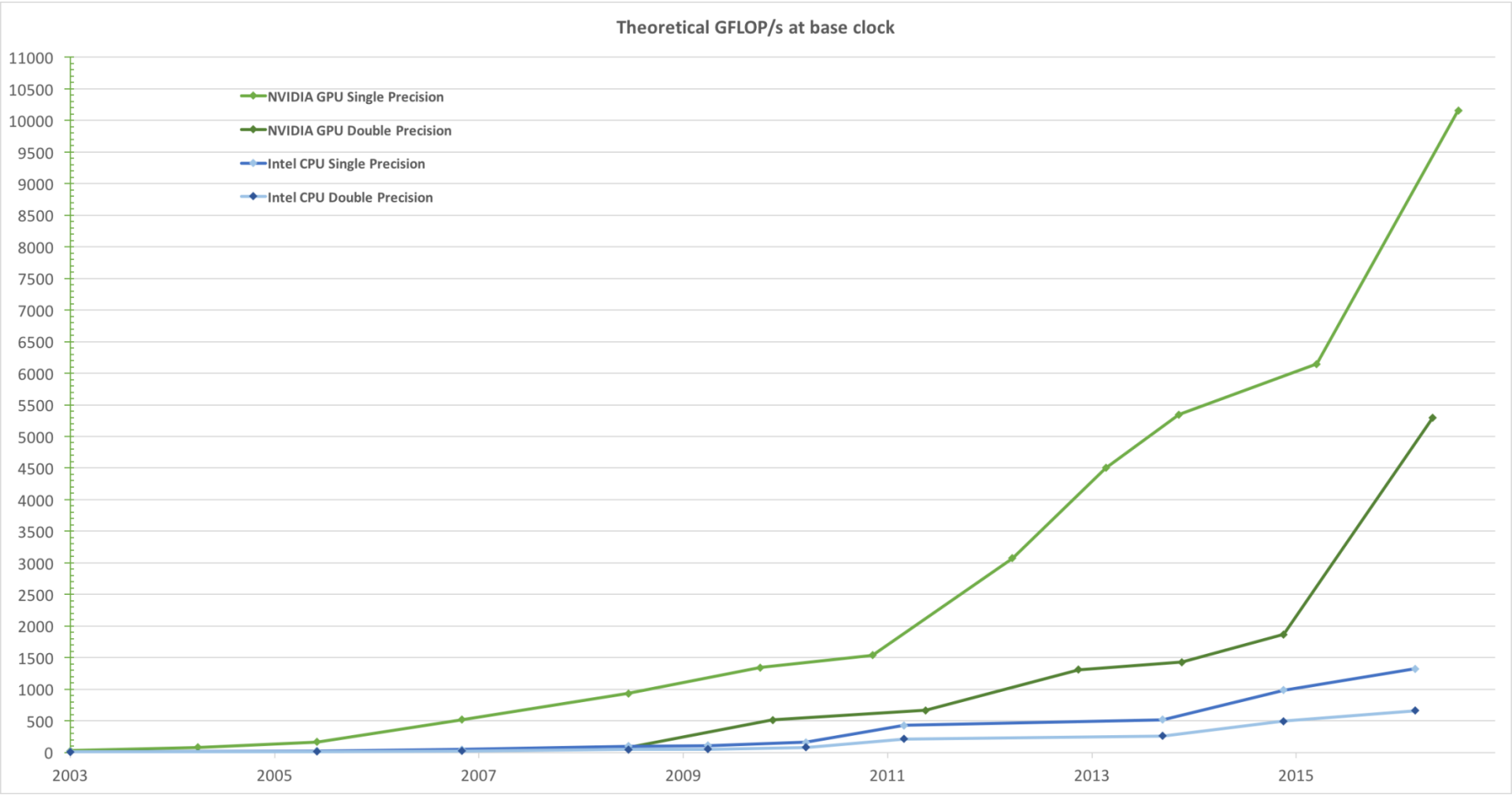
- GPUs evolved from graphics toward general purpose **data-parallel** workloads
- GPUs are commodity devices, omnipresent in modern computers (**millions** sold per **week**)
- **Massively parallel hardware**, well suited to **throughput-oriented** workloads, **streaming** data far too large for CPU caches
- Programming tools allow software to be written in various dialects of familiar C/C++/Fortran and integrated into legacy software
- GPU algorithms are often multicore-friendly due to attention paid to **data locality** and **data-parallel work decomposition**

What Makes GPUs Compelling?

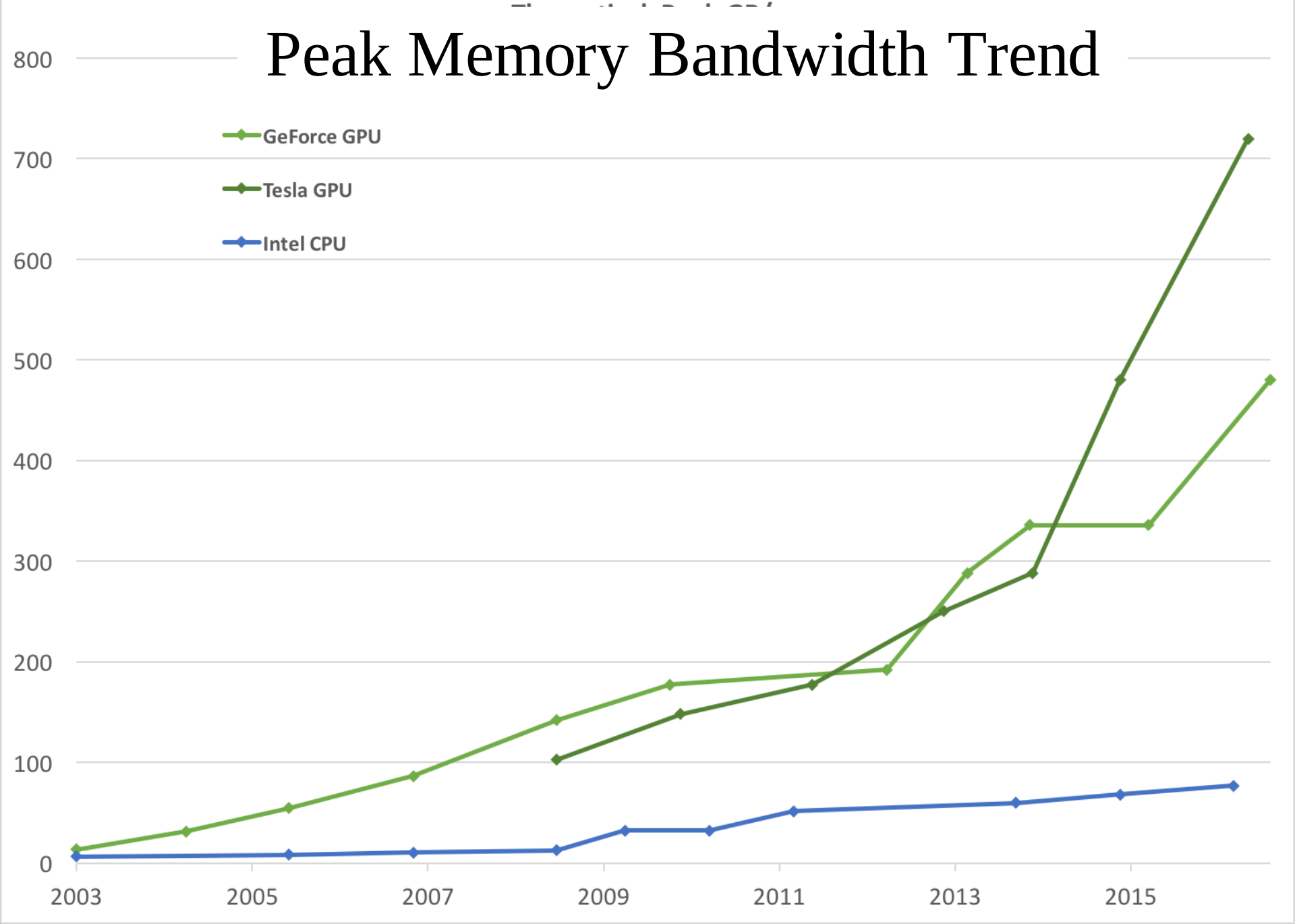
- **Massively parallel** hardware architecture:
  - Tens of wide SIMD-oriented stream processing compute units ("SMs" in NVIDIA nomenclature)
  - **Tens of thousands of threads running on thousands of ALUs, special fctn units**
  - **Large register files, fast on-chip and die-stacked memory systems**
- Example: NVIDIA Tesla V100 (Volta) **Peak** Perf:
  - **7.5 TFLOPS FP64, 15 TFLOPS FP32**
  - **120 TFLOPS Tensor unit (FP16/FP32 mix)**
  - **900 GB/sec memory bandwidth (HBM2)**

http://www.nvidia.com/object/volta-architecture-whitepaper.html

Peak Arithmetic Performance Trend



Peak Memory Bandwidth Trend



Evolution of GPUs Over Multiple Generations

Table 1. Comparison of NVIDIA Tesla GPUs

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
SMs	15	24	56	80
TPCs	15	24	28	40
FP32 Cores / SM	192	128	64	64
FP32 Cores / GPU	2880	3072	3584	5120
FP64 Cores / SM	64	4	32	32
FP64 Cores / GPU	960	96	1792	2560
Peak FP32 TFLOPS[1]	5	6.8	10.6	15
Peak FP64 TFLOPS[1]	1.7	.21	5.3	7.5
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size / SM	16 KB/32 KB/48 KB	96 KB	64 KB	Configurable up to 96 KB
Register File Size / SM	256 KB	256 KB	256 KB	256KB
Register File Size / GPU	3840 KB	6144 KB	14336 KB	20480 KB

http://www.nvidia.com/object/volta-architecture-whitepaper.html

Other Benefits of GPUs

- **2019: 22 of the top of 25 Green500 systems are GPU-accelerated (Tesla V100 or P100)**
  - **Increased GFLOPS/watt power efficiency**
  - Increased compute power per unit volume
- Desktop workstations can incorporate the **same types of GPUs** found in clouds, clusters, and supercomputers
- GPUs can be upgraded without new OS version, license fees, etc.

Sounds Great! What Don't GPUs Do?

- GPUs don't accelerate serial code…
- GPUs don't run your operating system…you still need a CPU for that…
- GPUs don't accelerate your InfiniBand card…
- GPUs don't make disk I/O faster…

…and…

- **GPUs don't make Amdahl's Law magically go away…**

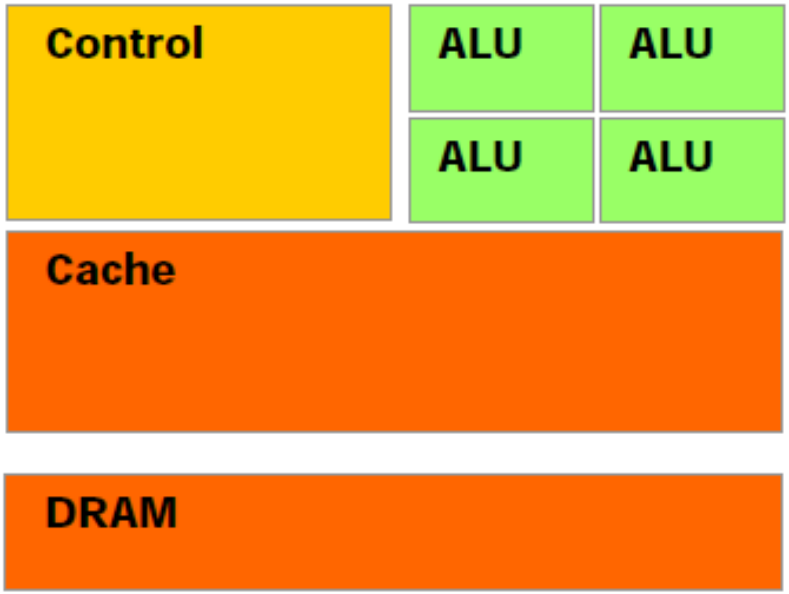
Heterogeneous Computing

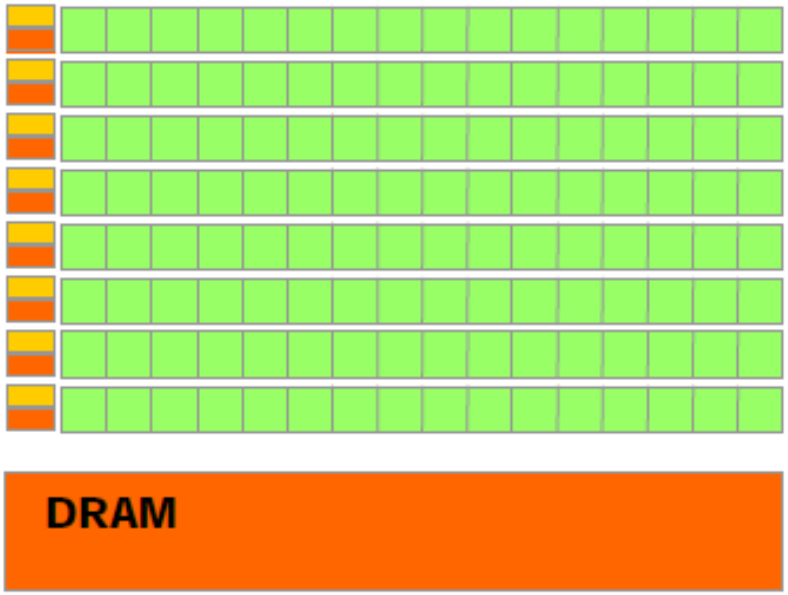
- Use processors with **complementary capabilities** for **best overall performance**
- GPUs of today are effective accelerators that depend on the "host" system for OS and resource management, I/O, etc…
- GPU-accelerated programs are therefore programs that run on **"heterogeneous computing systems"** consisting of a mix of processors (at least CPU+GPU)

Complementarity of Typical CPU and GPU Hardware Architectures

CPU: Cache heavy, **low latency**, per-thread performance, small core counts

GPU: ALU heavy, massively parallel, **throughput-oriented**





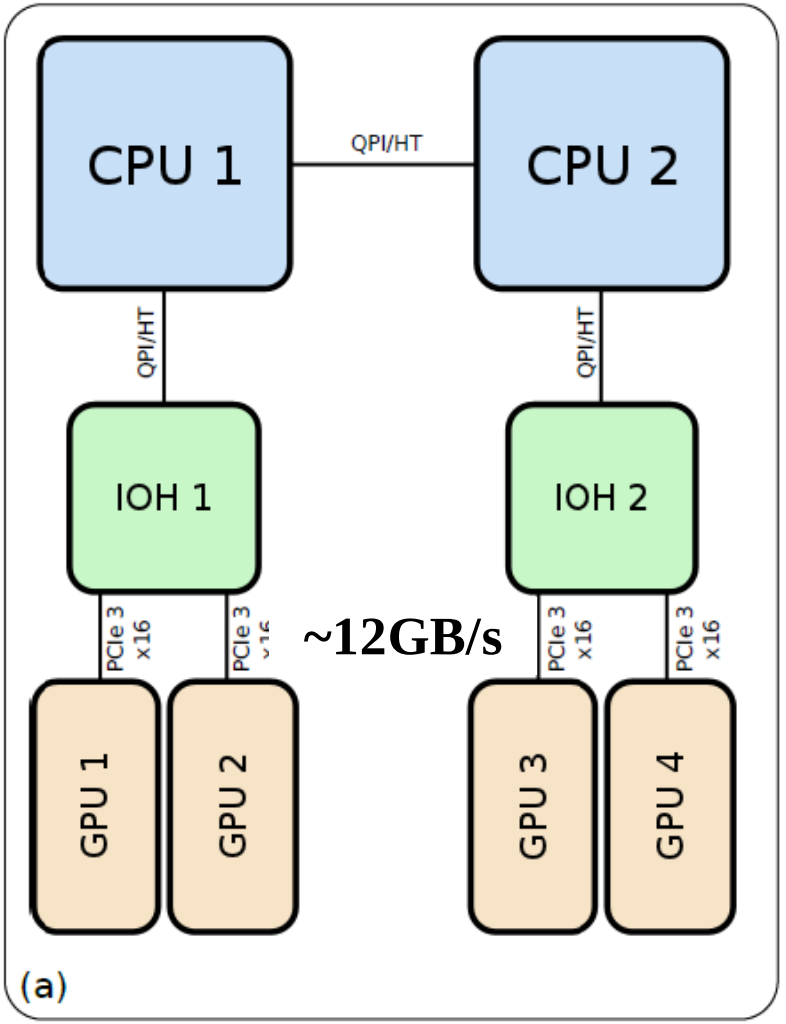
Exemplary Heterogeneous Computing Challenges

- **Tuning, adapting, or developing software for multiple processor types**
- Decomposition of problem(s) and load balancing work across heterogeneous resources for **best overall performance and work-efficiency**
- **Managing data placement** in disjoint memory systems with varying performance attributes
- **Transferring data** between processors, memory systems, interconnect, and I/O devices
- …

Heterogeneous Compute Node

- Dense **PCIe-based** multi-GPU compute node
- Application would **ideally exploit all** of the CPU, GPU, and I/O resources **concurrently**…

  (I/O devs not shown)



Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations
Michael J. Hallock, John E. Stone, Elijah Roberts, Corey Fry, and Zaida Luthey-Schulten.
Journal of Parallel Computing, 40:86-99, 2014.
http://dx.doi.org/10.1016/j.parco.2014.03.009

Major Approaches For Programming Hybrid Architectures

- Use **drop-in libraries** in place of CPU-only libraries
  - **Little or no code development**
  - Examples: MAGMA, BLAS-variants, FFT libraries, etc.
  - **Speedups limited by Amdahl's Law** and overheads associated with data movement between CPUs and GPU accelerators
- Generate accelerator code as a variant of CPU source, e.g. using OpenMP and **OpenACC directives**, and similar
- Write **lower-level** accelerator-specific code, e.g. using **CUDA, OpenCL**, other approaches

Simplified GPU-Accelerated Application Adaptation and Development Cycle

1. Use **drop-in GPU libraries**, e.g. BLAS, FFT, …
2. Profile application, **identify** opportunities for **massive data-parallelism**
3. **Migrate** well-suited **data-parallel work to GPUs**
   - **Run data-parallel work, e.g. loop nests on GPUs**
   - **Exploit high bandwidth memory systems**
   - Exploit massively parallel arithmetic hardware
   - Minimize host-GPU data transfers
4. Go back to step 2…
   - **Observe Amdahl's Law, adjust CPU-GPU workloads…**

What Runs on a GPU?

- GPUs run **data-parallel programs** called **"kernels"**
- GPUs are managed by host CPU thread(s):
  - Create a CUDA / OpenCL / OpenACC context
  - Manage GPU memory allocations/properties
  - Host-GPU and GPU-GPU (peer to peer) transfers
  - Launch GPU kernels
  - Query GPU status
  - Handle runtime errors

How Do I Write GPU Kernels?

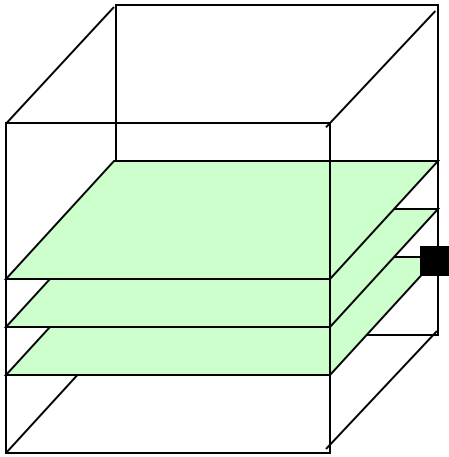
- Directive-based parallelism (OpenACC):
  - Annotate existing source code loop nests with directives that allow a compiler to automatically generate data-parallel kernels
  - Same source code targets multiple processors
- Explicit parallelism (CUDA, OpenCL)
  - Write data parallel kernels, explicitly map range of independent work items to GPU threads and groups
  - Explicit control over specialized on-chip memory systems, low-level parallel synchronization, reductions

Process for Writing CUDA **Kernels**

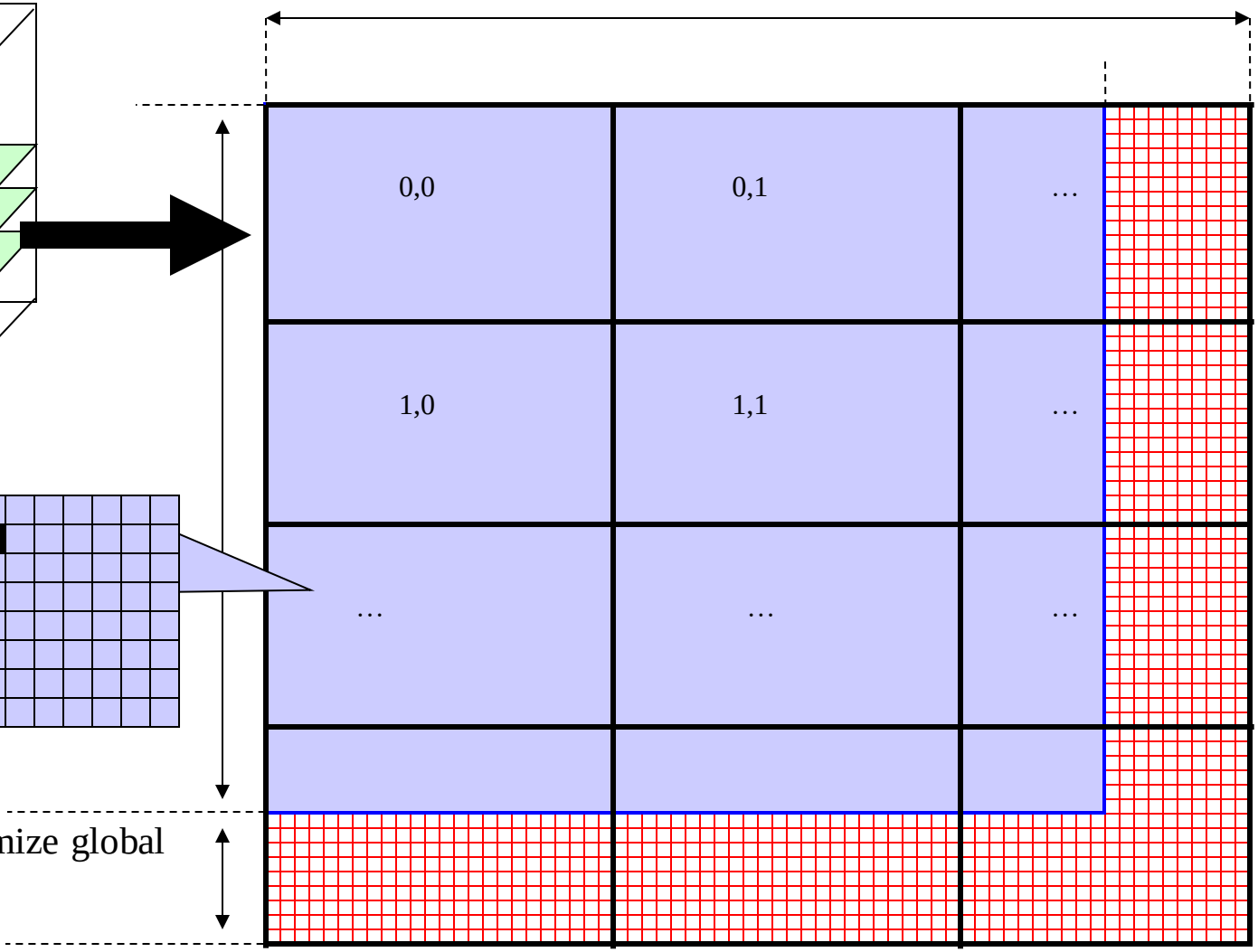
- **Data-parallel loop nests** are unrolled into a **large batch** of **independent** work items that can **execute concurrently**
- **Work** items are **mapped onto GPU hardware threads** using multidimensional grids and blocks of threads that execute on stream processing units (SMs)
- Programmer manages data placement in GPU memory systems, access patterns, and data dependencies

CUDA Grid, Block, Thread Decomposition

1-D, 2-D, or 3-D Computational Domain



1-D, 2-D, or 3-D **Grid** of **Thread Blocks**:



1-D, 2-D, 3-D **thread block**:

Padding arrays can optimize global memory performance

Overview of Throughput-Oriented GPU Hardware Architecture

- GPUs have small on-chip caches
- Main memory latency (several hundred clock cycles!) is tolerated through hardware multithreading – **overlap memory transfer latency with execution of other work**
- When a GPU thread stalls on a memory operation, the hardware **immediately switches** context to a ready thread
- Effective latency hiding requires saturating the GPU with lots of work – **tens of thousands** of **independent** work items

Avoid Output Conflicts, Conversion of Scatter to Gather

- Many CPU codes contain algorithms that **"scatter"** outputs to memory, to reduce arithmetic
- Scattered output can create bottlenecks for GPU performance due **write conflicts** among hundreds or **thousands of threads**
- On the GPU, it is often better to:
  - do **more arithmetic**, in exchange for **regularized output** memory write patterns
  - **convert "scatter" algorithms to "gather" approaches**
  - Use **data "privatization"** to reduce the scope of potentially conflicting outputs, and to leverage special **on-chip memory systems** and data reduction instructions

GPU Technology Conference Presentations

See the latest announcements about GPU hardware, libraries, and programming tools

- https://www.nvidia.com/en-us/gtc/
- https://www.nvidia.com/en-us/gtc/topics/

Questions?

Scaling in a Heterogeneous Environment with GPUs
CUDA Programming: Fundamental Abstractions

John E. Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

http://www.ks.uiuc.edu/~johns/

Petascale Computing Institute,

National Center for Supercomputing Applications,

University of Illinois at Urbana-Champaign

An Approach to Writing CUDA Kernels

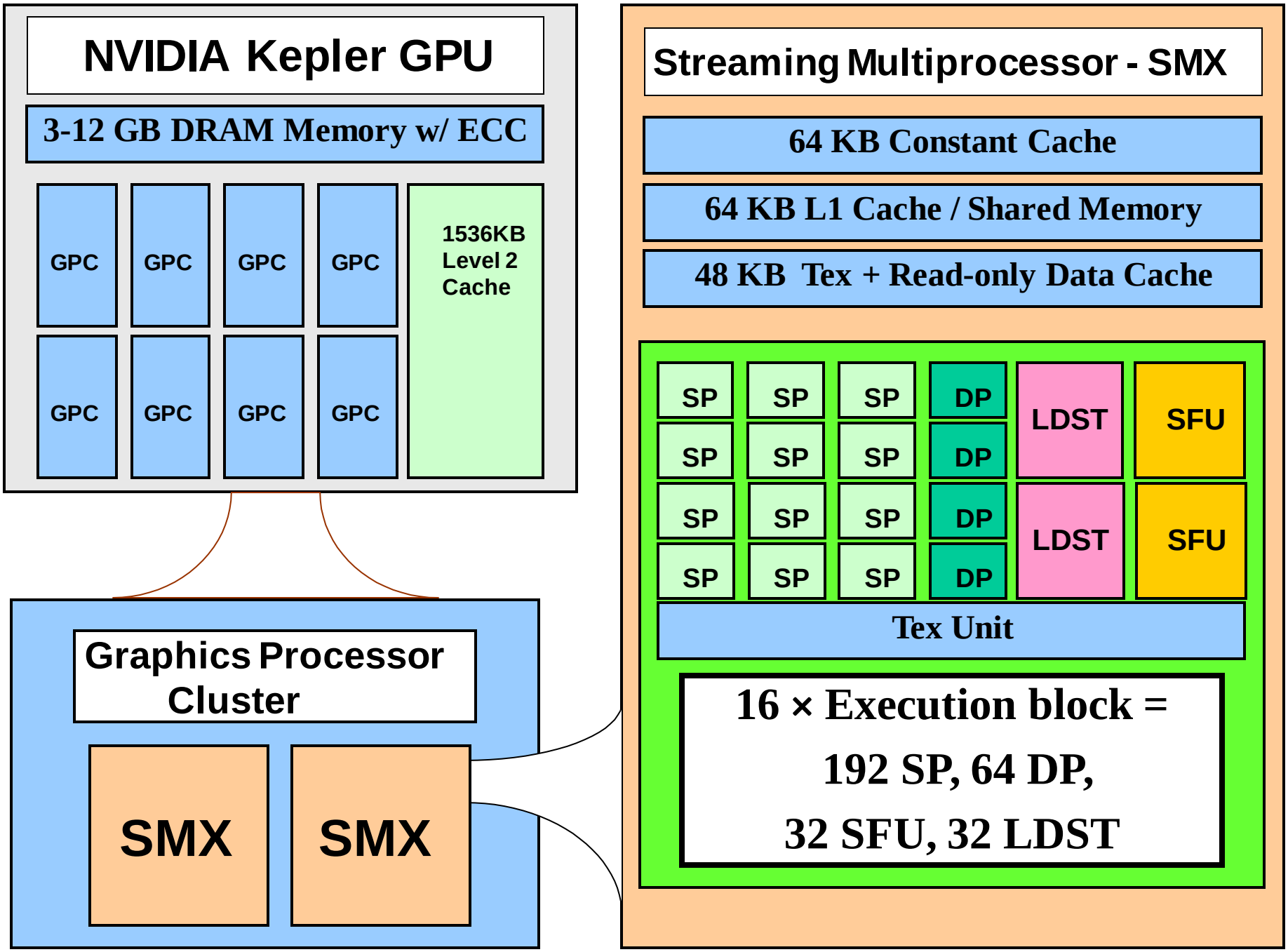
- Find an algorithm that can expose **substantial parallelism**, we'll ultimately need thousands of independent threads…
- Identify **most appropriate** GPU memory systems to store data used by kernel, considering its **access pattern**
- Can trade-offs be made to **exchange arithmetic for fewer memory accesses or more parallelism**?
  - Though **counterintuitive**, some past successes resulted from this
  - "Brute force" methods that expose significant parallelism do surprisingly well on GPUs
- Analyze the real-world use cases for the problem and design or select a specialized kernel for the problem sizes that will be heavily used

Getting Performance From GPUs

- Don't worry about counting arithmetic operations until you have nothing else left to do…
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration when programming GPUs is **accessing memory efficiently**, and storing operands in the **most appropriate memory system** according to data size and access pattern

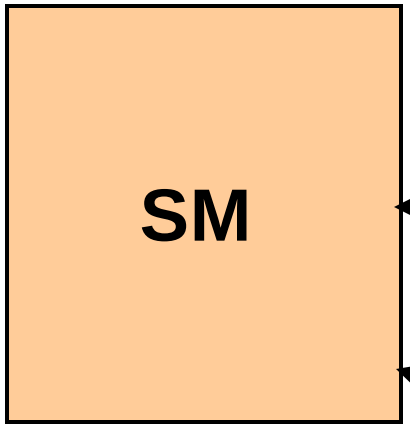
CUDA Work Abstraction

- Work is expressed as a multidimensional array of independent work items called **"threads"** – not the same thing as a CPU thread
- CUDA **Kernels** can be thought of as telling a GPU to compute **all iterations** of a set of nested loops **concurrently**
- Threads are dynamically scheduled onto hardware according to a hierarchy of thread groupings

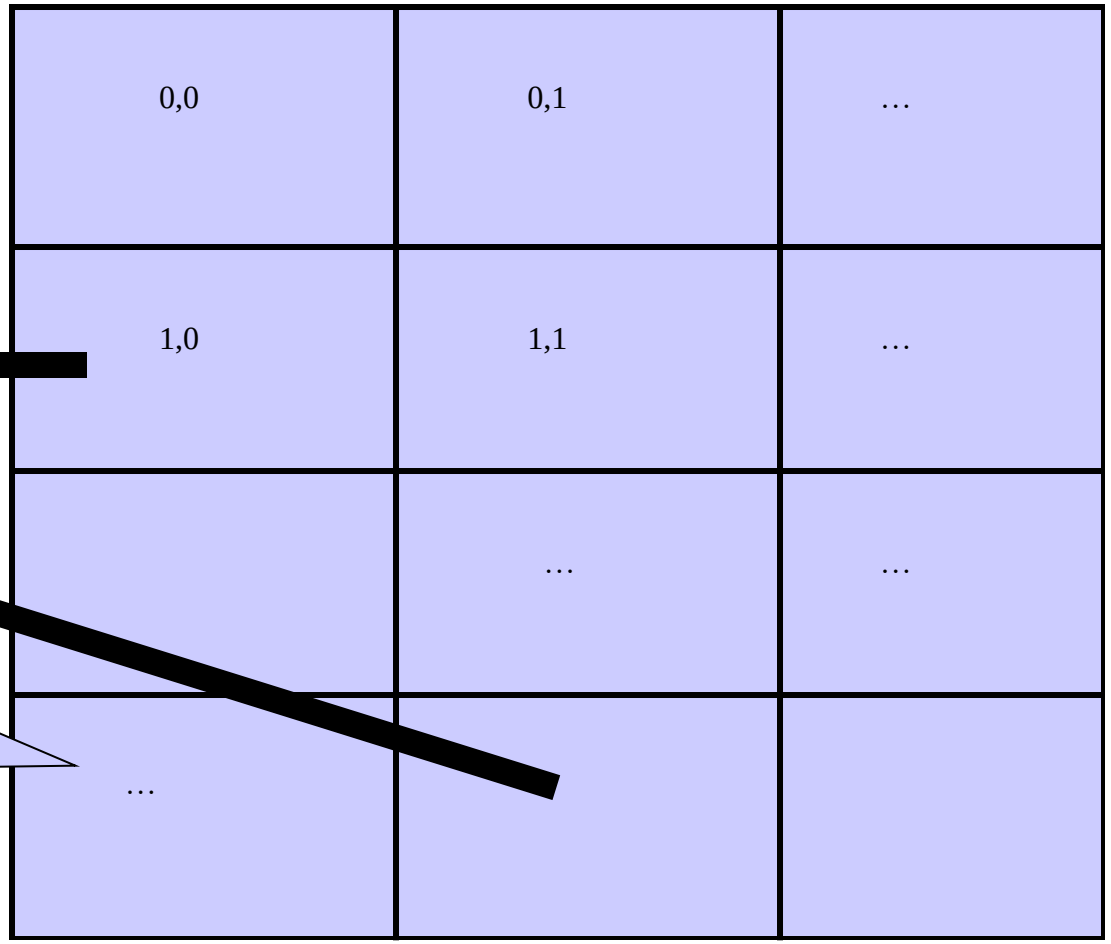

NVIDIA Kepler GPU
3-12 GB DRAM Memory w/ ECC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
1536KB Level 2 Cache
Graphics Processor Cluster
SMX
SMX
Streaming Multiprocessor - SMX
64 KB Constant Cache
64 KB L1 Cache / Shared Memory
48 KB Tex + Read-only Data Cache
SP
DP
LDST
SFU
Tex Unit
16 × Execution block =
192 SP, 64 DP,
32 SFU, 32 LDST

CUDA Work Abstractions: Grids, Thread Blocks, Threads

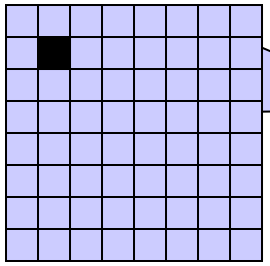
Thread blocks are scheduled onto pool of GPU SMs…



1-D, 2-D, or 3-D **Grid** of **Thread Blocks**:



1-D, 2-D, 3-D thread block:



Basic CUDA Kernel Syntax

- **__global__** tells the **nvcc** compiler to generate kernel code that can run on the GPU:

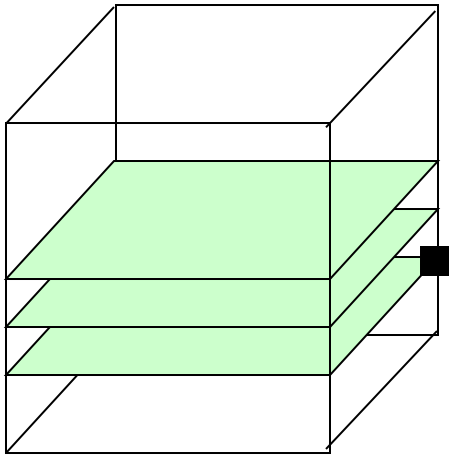
```
__global__ void MyKernel(int myparams, ...) {
```

- **Read-only kernel variables:**
  - **blockIdx.x** thread's index within its block
  - **blockDim.x** block's index within the grid
  - **gridDim.x** number of blocks in the grid
- Kernels can be launched by host or GPU itself:

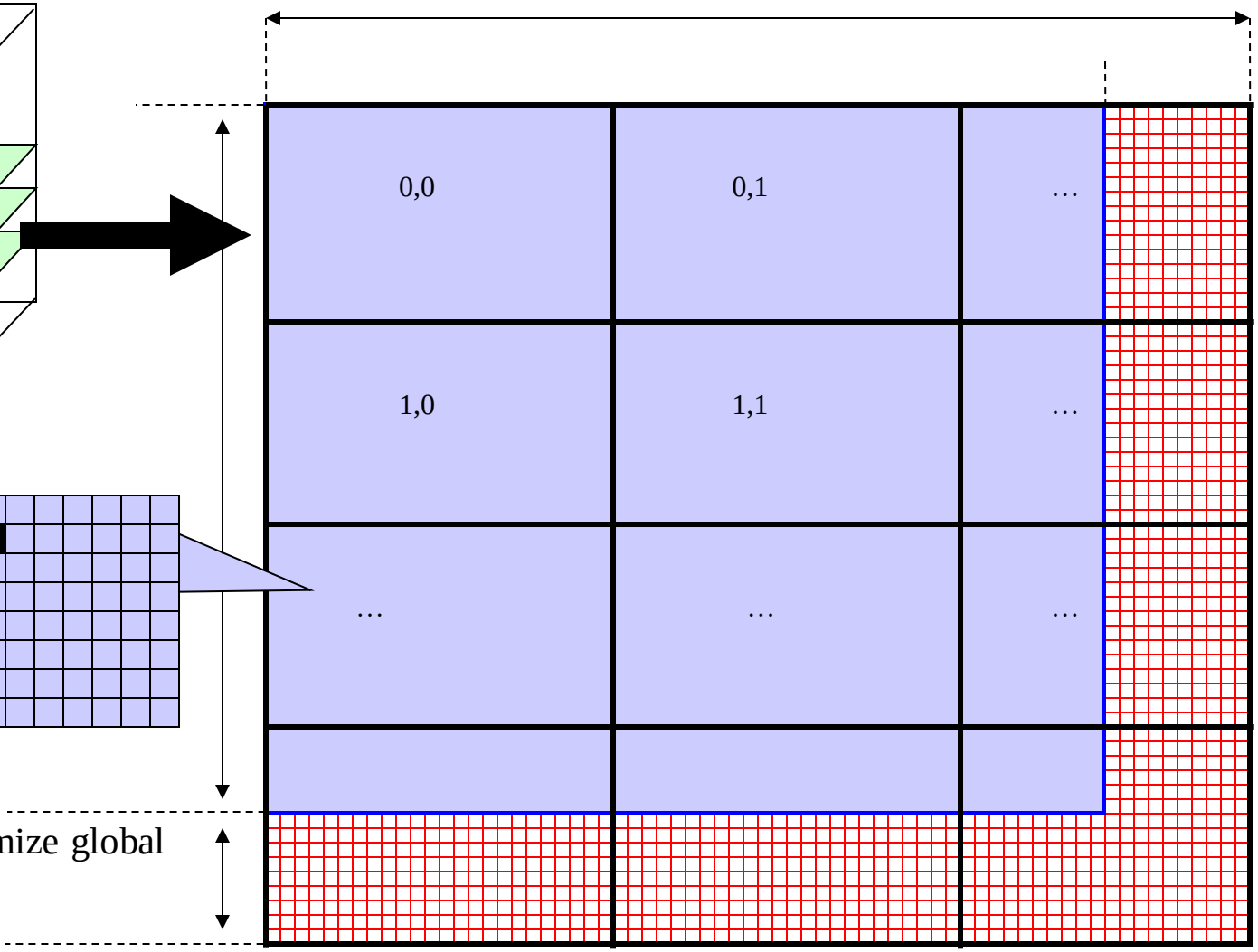
```
MyKernel<<<GridDims, BlockDims>>>(parms);
```

CUDA Grid, Block, Thread Decomposition

1-D, 2-D, or 3-D Computational Domain



1-D, 2-D, or 3-D **Grid** of **Thread Blocks**:

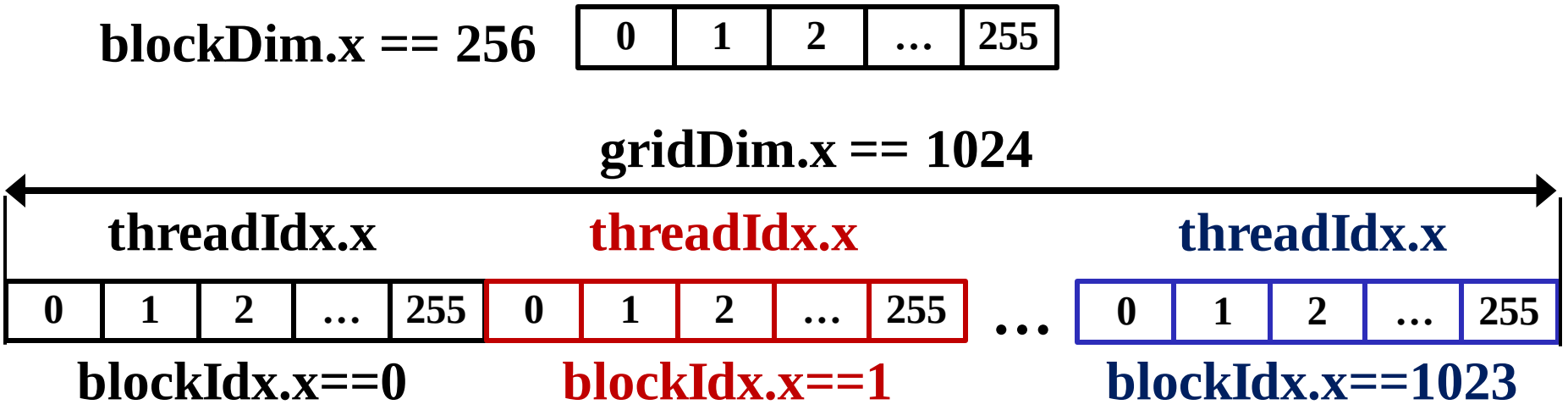


1-D, 2-D, 3-D **thread block**:

Padding arrays can optimize global memory performance

Indexing Work

- Within a CUDA kernel:
  - Grid: gridDim.[xyz]
  - Block: blockDim.[xyz] and blockIdx.[xyz]
  - Thread: threadIdx.[xyz]

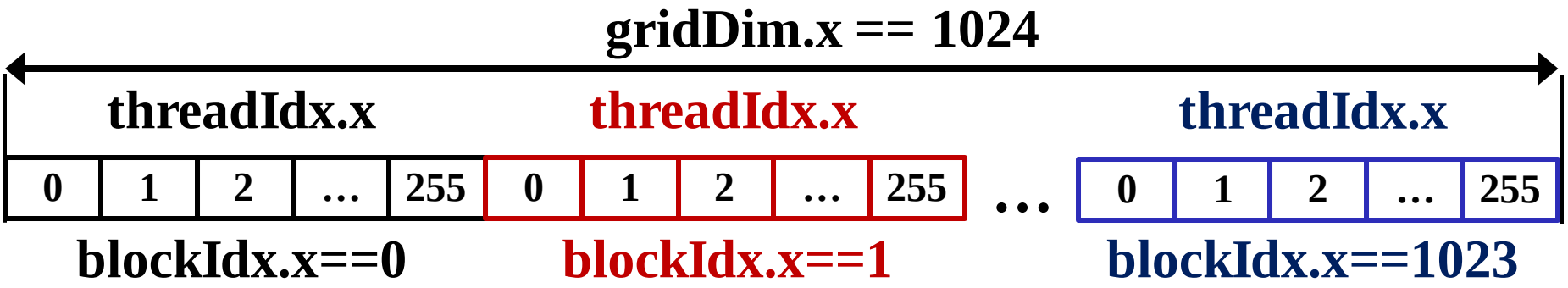


GlobalThreadIndex= **(blockIdx.x * blockDim.x) + threadIdx.x;**

Indexing Work

- Example CUDA kernel with 1-D Indexing:

```
__global__ void cuda_add(float *c, float *a, float *b) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  c[idx] = a[idx] + b[idx];
}
```



What if Work Size Isn't an Integer Multiple of the Thread Block Size?

- Grid size must be rounded up to account for final partially-full thread block:

  GridSZ = (N + BlockSZ - 1) / BlockSZ

- Threads must check if they are "in bounds":

```
__global__ void cuda_add(float *c, float *a, float *b, int N) {
  int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
  if (idx < N) {
    c[idx] = a[idx] + b[idx];
  }
}
```

Running a GPU kernel:

```
int sz = N * sizeof(float);
…
cudaMalloc((void**) &a_gpu, sz);
cudaMemcpy(a_gpu, a, sz, cudaMemcpyHostToDevice);
… // do the same for ‘b_gpu’, allocate ‘c_gpu’
int Bsz = 256; // 1-D thread block size
cuda_add<<<(N+Bsz-1)/Bsz, Bsz>>>(c, a, b);
cudaDeviceSynchronize(); // make CPU wait for completion
...
cudaMemcpy(c, c_gpu, sz, cudaMemcpyDeviceToHost);
cudaFree(a_gpu);
… // free ‘b_gpu’, and ‘c_gpu’…
```

CUDA Explicit GPU Memory Allocation and Data Transfer APIs

- cudaMalloc (void** devPtr, size_t size)
  - Create a GPU memory allocation
  - Works just like malloc() except it affects GPU memory
  - Pointer returned is a GPU virtual address that isn't directly accessible on the host, so memory must be populated with data using cudaMemcpy*() calls…
- cudaMemcpy (void* dst, const void* src, size_t count, cudaMemcpyKind kind)
  - Perform Host-GPU or GPU-GPU DMA transfer
  - The "kind" parameter tells the runtime which memory systems the source and destination addresses refer to
- cudaFree(void *)
  - Destroy any GPU memory allocation

CUDA Managed (Automated Transfers) GPU Memory Allocation API

- cudaMallocManaged (T** devPtr, size_t size)
  - GPU, CUDA driver, and runtime system manage the memory allocation, automatically performing host-GPU DMA transfers when needed as the host or device read/write memory
  - Reduces the need for explicit management of data transfers and frees programmers to focus on other aspects of GPU algorithm development unless/until there is a need to achieve performance that the automatic management approach cannot achieve by itself
  - Performs best on the latest GPU hardware
  - A Code Ninja can beat automated management, but it's an excellent way to do early development and prototyping
- cudaFree(void *)
  - Destroy any GPU memory allocation

Example CUDA Stream of Execution: Explicit Data Transfers

```
cudaMalloc((void**) &gin, insz);
cudaMalloc((void**) &gout, osz);
hin=malloc(insz);  hout=malloc(osz);  DoCPUInitStuff(hin);
cudaMemcpy(gin, hin, insz, cudaMemcpyHostToDevice);
MyKernel<<<GridSZ, BlockSZ>>>(gin, gout);
cudaMemcpy(hout, gout, osz, cudaMemcpyDeviceToHost);
free(hin);  free(hout);  DoCPUFinishStuff(hout)
cudaFree(gin);
cudaFree(gout);
```

Example CUDA Stream of Execution: Managed Memory, w/ Implicit Data Transfers

```
cudaMallocManaged((void**) &in, insz);
cudaMallocManaged((void**) &out, osz);
hin=malloc(insz); hout=malloc(osz); DoCPUInitStuff(in);
cudaMemcpy(gin, hin, insz, cudaMemcpyHostToDevice)
MyKernel<<<GridSZ, BlockSZ>>>(in, out);
cudaMemcpy(hout, gout, osz, cudaMemcpyDeviceToHost)
free(hin); free(hout); DoCPUFinishStuff(out)
cudaFree(in);
cudaFree(out);
```

Example CUDA Stream of Execution: Managed Memory, w/ Implicit Data Transfers

```
cudaMallocManaged((void**) &in, insz)
cudaMallocManaged((void**) &out, osz)
DoCPUInitStuff(in);
MyKernel<<<GridSZ, BlockSZ>>>(in, out);
cudaDeviceSynchronize(); // for Kepler/Maxwell GPUs
 DoCPUFinishStuff(out)
cudaFree(in);
cudaFree(out);
```

Additional Reference Materials:

- https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/
- http://docs.nvidia.com/cuda/cuda-c-programming-guide
- The CUDA Handbook: A Comprehensive Guide to GPU Programming
  - https://www.amazon.com/CUDA-Handbook-Comprehensive-Guide-Programming/dp/0321809467
- Programming Massively Parallel Processors: A Hands-on Approach (Third Edition)
  - https://www.amazon.com/Programming-Massively-Parallel-Processors-Hands/dp/0128119861

Questions?

Scaling in a Heterogeneous Environment with GPUs
CUDA Programming 2:
GPU Thread Execution and Memory Systems

John E. Stone

Theoretical and Computational Biophysics Group

Beckman Institute for Advanced Science and Technology

University of Illinois at Urbana-Champaign

http://www.ks.uiuc.edu/~johns/

Scaling to Petascale Institute,

National Center for Supercomputing Applications,

University of Illinois at Urbana-Champaign

Getting Performance From GPUs

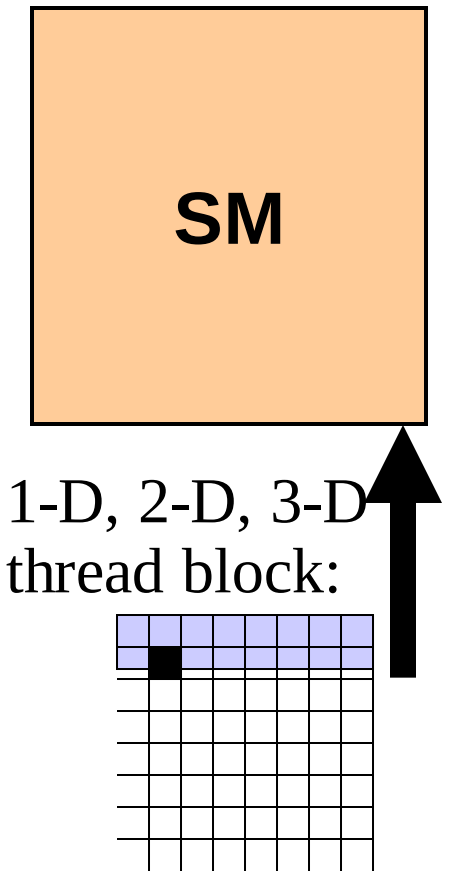
- Don't worry (much) about counting arithmetic operations…at least until you have nothing else left to do
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration when programming GPUs is **accessing memory efficiently**, and storing operands in the **most appropriate memory system** according to data size and access pattern

GPU Thread Block Execution

- Thread blocks are decomposed onto hardware in **32-thread "warps"**
- Hardware execution is scheduled in units of **warps** – an SM can execute warps from several thread blocks
- Warps run in SIMD-style execution:
  - **All threads execute the same instruction in lock-step**
  - **If one thread stalls, the entire warp stalls…**
  - **A branch taken by any thread has to be taken by all threads...**

(divergence is undesirable)

Thread blocks are multiplexed onto pool of GPU SMs…



GPU On-Board Global Memory

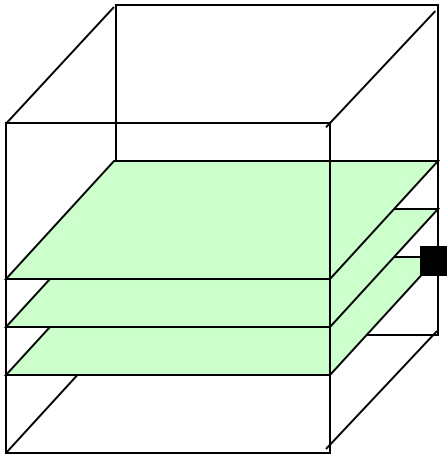
- GPU arithmetic rates dwarf memory bandwidth
- For old Kepler K40 hardware:
  - ~4.3 FP32 TFLOPS vs. ~288 GB/sec
  - The ratio is roughly **60 FLOPS per memory reference** for single-precision floating point
- Peak performance achieved with **"coalesced"** memory access patterns – patterns that result in a single hardware memory transaction for a SIMD **"warp" – a contiguous group of 32 threads**

Memory Coalescing

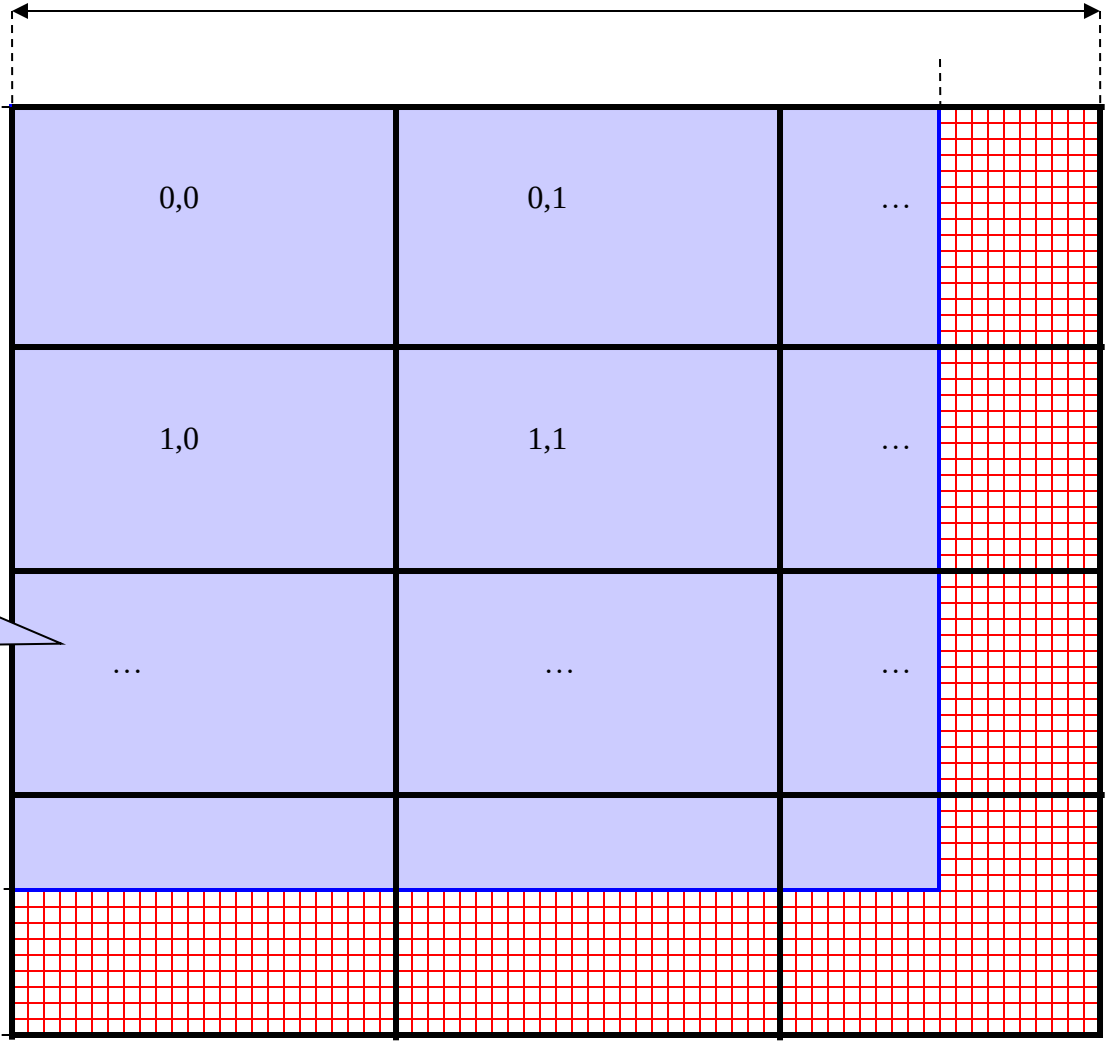
- Oversimplified explanation:
  - Threads in a warp perform a read/write operation that can be serviced in a **single hardware transaction**
  - New GPUs are much more flexible than old ones
  - If all threads in a warp read from a contiguous region that's 32 items of 4, 8, or 16 bytes in size, that's an example of a coalesced access
  - Multiple threads reading the same data are handled by a **hardware broadcast** (can provide memory bandwidth amplification when exploited in a kernel)
  - Writes are similar, but multiple writes to the same location yield undefined results

CUDA Grid/Block/Thread **Padding**

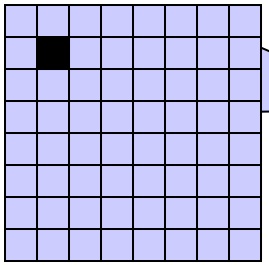
1-D, 2-D, or 3-D Computational Domain



1-D, 2-D, or 3-D Grid of Thread Blocks:



1-D, 2-D, 3-D thread block:



Padding arrays out to full blocks optimizes global memory performance by guaranteeing memory coalescing

Using the CPU to Optimize GPU Performance

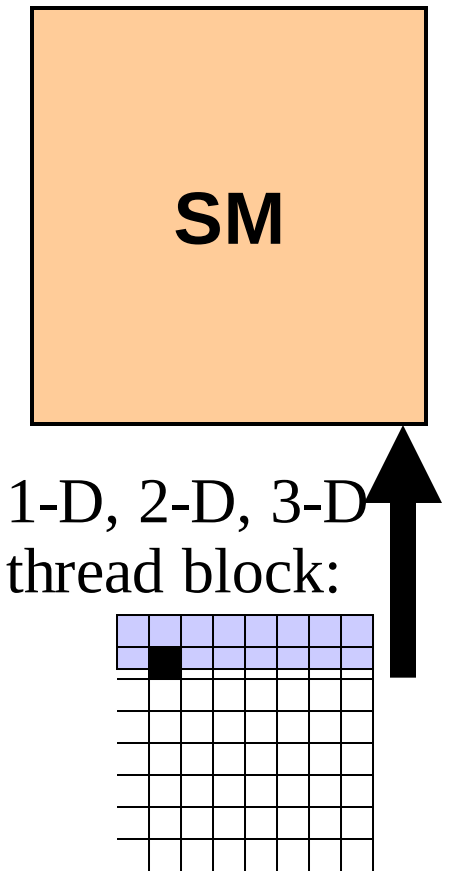
- GPU performs best when the work evenly divides into the number of threads/processing units
- Optimization strategy to consider:
  - Use the CPU to ***"regularize"*** the GPU workload
  - Maybe use **fixed-size** bin data structures, with "empty" slots skipped or producing zeroed out results
  - Handle exceptional or irregular work units on the CPU; GPU processes the bulk of the work concurrently
  - On average, the GPU is kept work-efficient and highly occupied, attaining a high fraction of peak performance

GPU Thread Block Execution

- Thread blocks are decomposed onto hardware in **32-thread "warps"**
- Hardware execution is scheduled in units of **warps** – an SM can execute warps from several thread blocks
- Warps run in SIMD-style execution:
  - **All threads execute the same instruction in lock-step**
  - **If one thread stalls, the entire warp stalls…**
  - **A branch taken by any thread has to be taken by all threads...**

(divergence is undesirable)

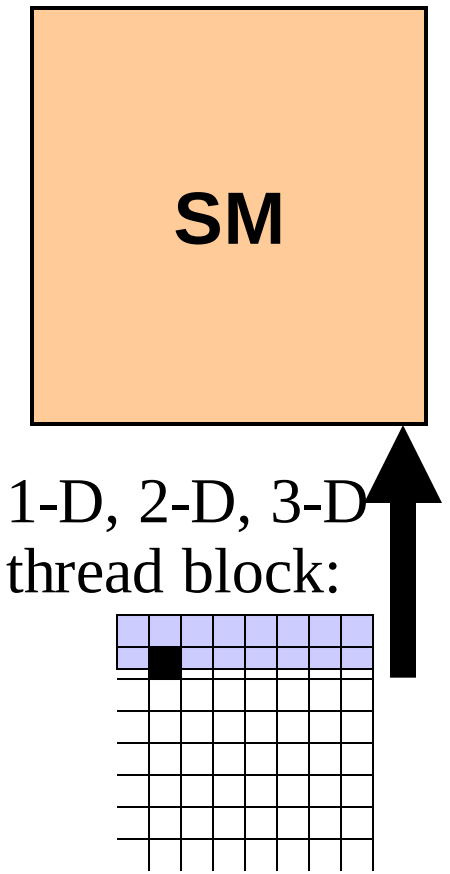
Thread blocks are multiplexed onto pool of GPU SMs…



GPU Warp Branch Divergence

- Branch divergence: when not all threads take the same branch, the entire warp has to **execute both sides of the branch**
- GPU hardware blocks memory writes from disabled threads in the "if then" branch, then inverts all thread enable states and runs the "else" branch
- GPU hardware detects warp reconvergence and then runs normally...
- Divergence is an issue for all SIMD hardware designs…
- GPUs benefit from a completely **hardware-based** implementation

Thread blocks are multiplexed onto pool of GPU SMs…



GPU Thread "Occupancy"

- GPU hardware designed to **oversubscribe ALUs with lots of threads**, thereby **tolerating** very long memory **latencies** without large on-chip caches
- Occupancy refers to the degree to which the GPUs warp schedulers are "full" of threads
- High occupancy often (but not always) improves latency hiding, which is often (but not always) better for performance
- Sometimes it is possible to achieve good performance even with relatively low occupancy, via schemes that reuse registers, increase work-efficiency, instruction-level parallelism, etc.
- Occupancy is limited by a kernel's register use, shared memory requirement, block size, and the available number of blocks in a grid – Explore **CUDA Occupancy Calculator Spreadsheet**!

Off-GPU Memory Accesses

- Direct access or transfer to/from host memory or peer GPU memory
  - Zero-copy behavior for accesses within kernel
  - Accesses become PCIe transactions
  - Overlap kernel execution with memory accesses
    - faster if accesses are coalesced
    - slower if not coalesced or multiple writes or multiple reads that miss the small GPU caches
- Host-mapped memory
  - cudaHostAlloc() – allocate GPU-accessible host memory

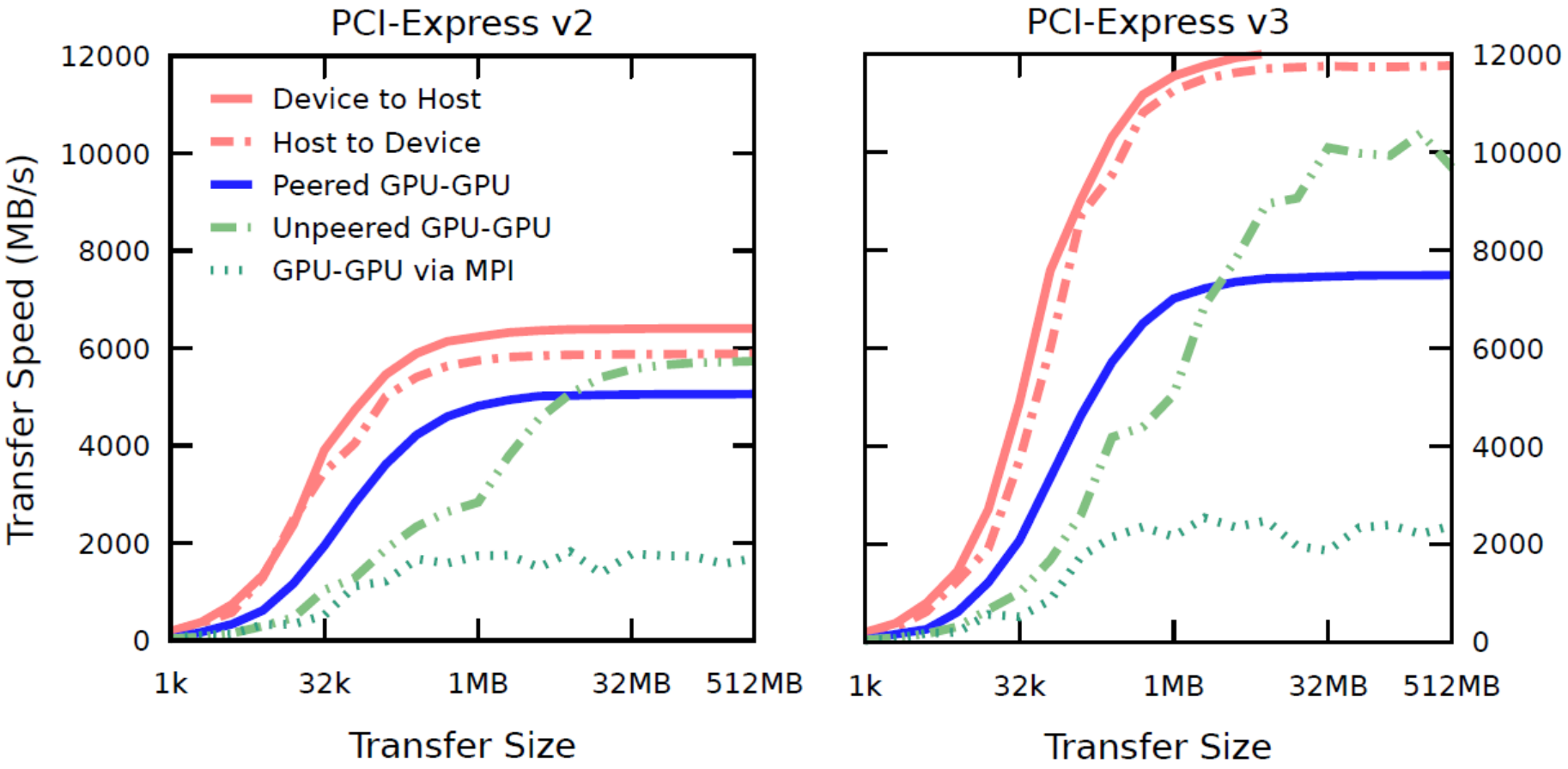
Off-GPU Memory Accesses

- Unified Virtual Addressing (UVA)
  - CUDA driver ensures that all GPUs in the system use unique non-overlapping ranges of virtual addresses which are also distinct from host VAs
  - CUDA decodes target memory space automatically from the pointer
  - Greatly simplifies code for:
    - GPU accesses to mapped host memory
    - Peer-to-Peer GPU accesses/transfers
    - MPI accesses to GPU memory buffers
- Leads toward Unified Virtual Memory (UVM)

Page Locked (Pinned) Host Memory

- Allocates host memory that is marked unmoveable in the OS VM system, so hardware can safely DMA to/from it
- Enables Host-GPU DMA transfers that approach full PCIe bandwidth:
  - PCIe 2.x 6 GB/s
  - PCIe 3.x 12 GB/s
- Enables full overlap of Host-GPU DMA and simultaneous kernel execution
- Enables simultaneous bidirectional DMAs to/from host

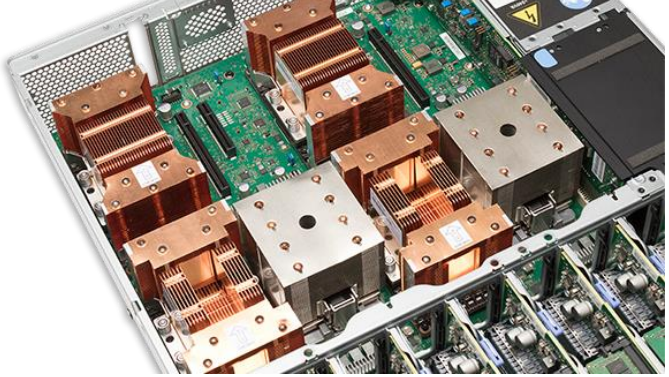
GPU PCI-Express DMA

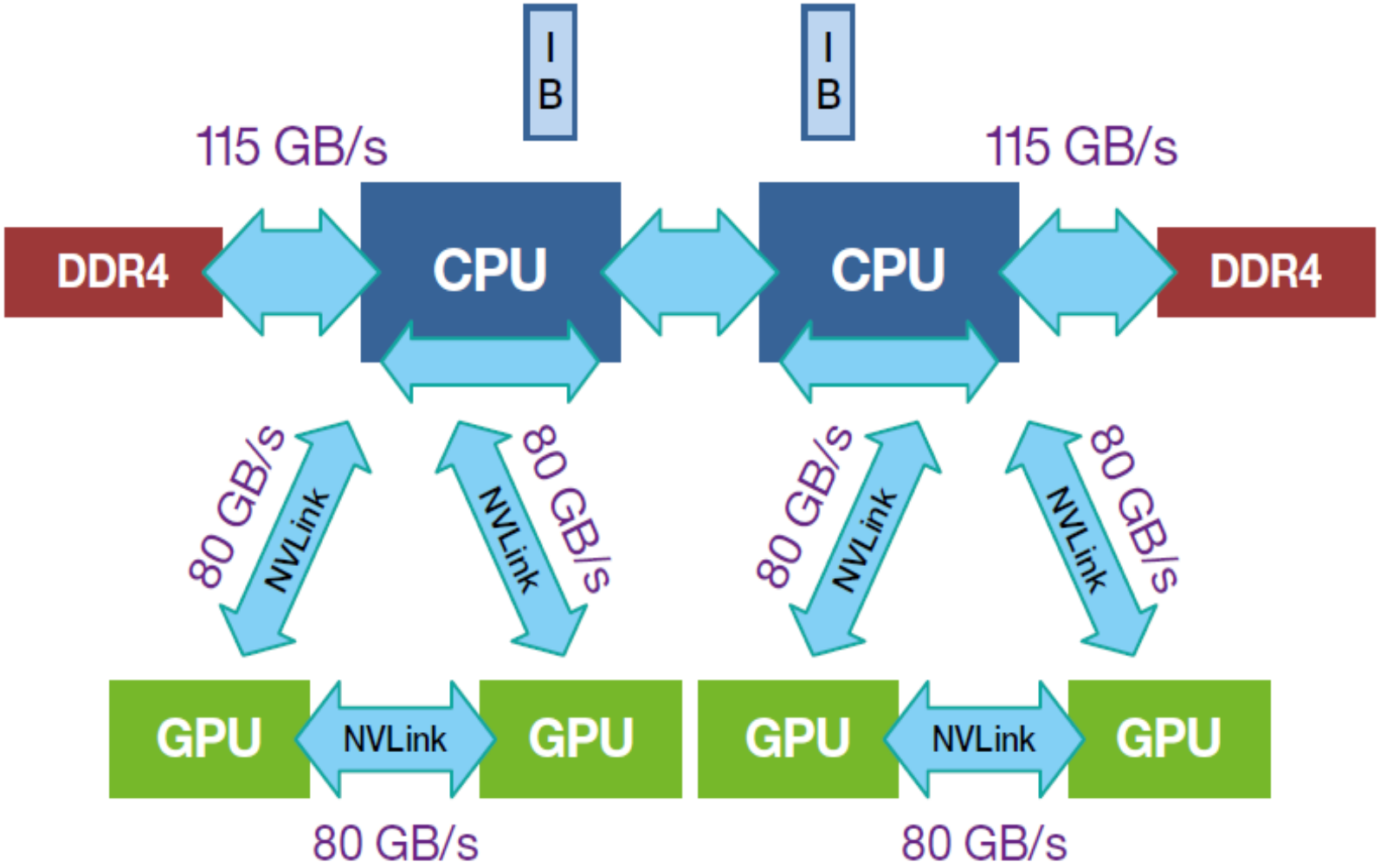


Simulation of reaction diffusion processes over biologically relevant size and time scales using multi-GPU workstations
Michael J. Hallock, John E. Stone, Elijah Roberts, Corey Fry, and Zaida Luthey-Schulten.
Journal of Parallel Computing, 40:86-99, 2014.
http://dx.doi.org/10.1016/j.parco.2014.03.009

IBM S822LC w/ **NVLink 1 .0**

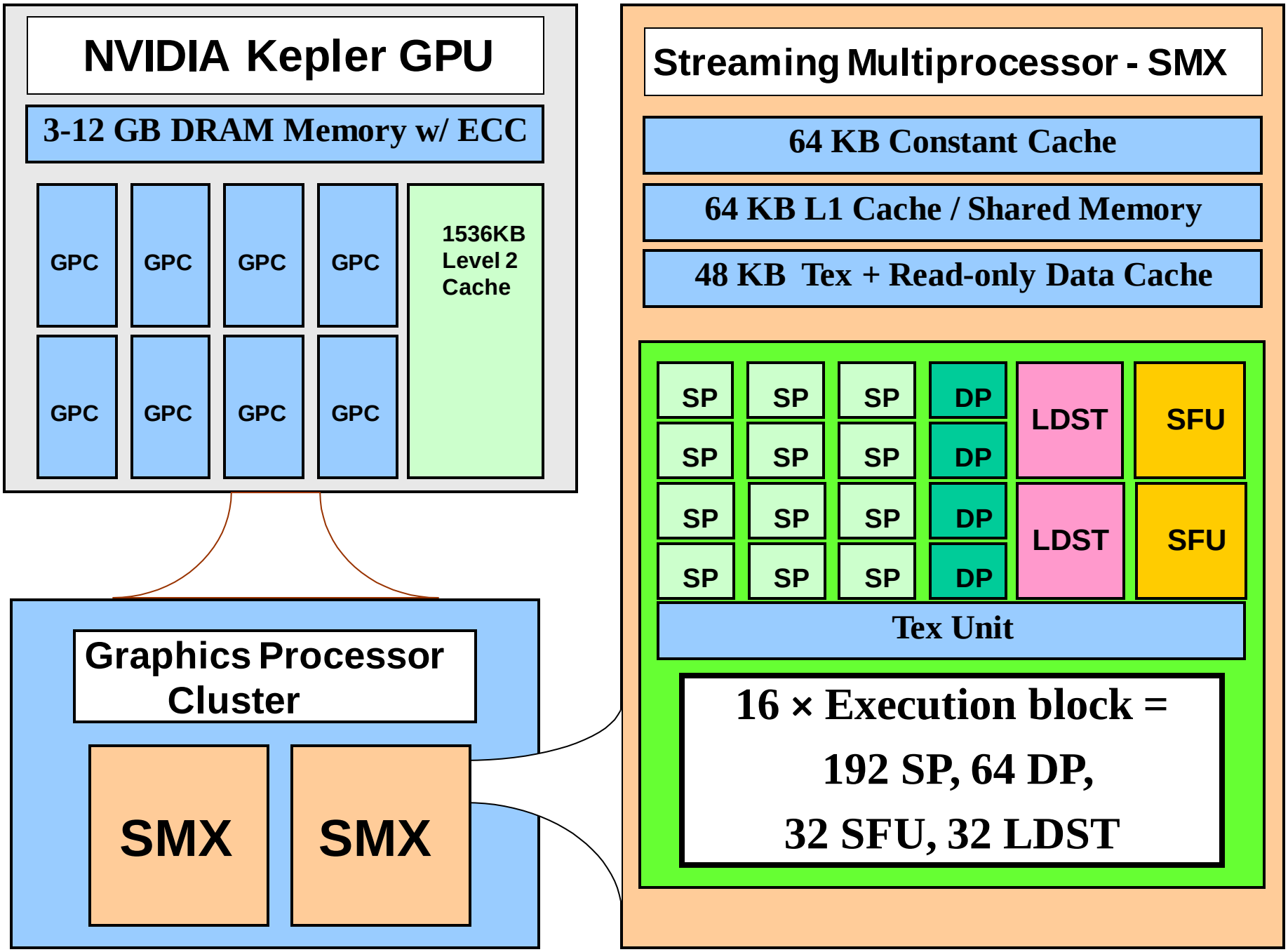
“Minsky”





GPU On-Chip Memory Systems

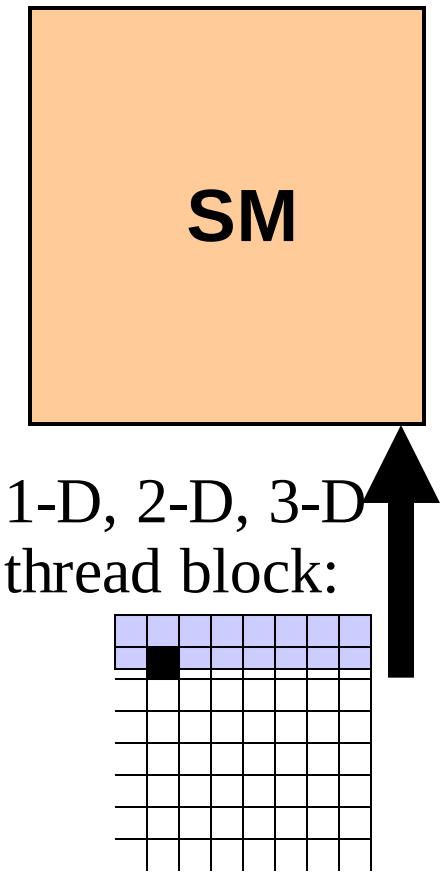
- GPU arithmetic rates dwarf global memory bandwidth
- GPUs include multiple fast **on-chip** memories to help **narrow the gap**:
  - **Registers**
  - Constant memory (64KB)
  - **Shared memory (64KB / 48KB / 16KB, varies…)**
  - Read-only data cache / Texture cache (~48KB)
    - Hardware-assisted 1-D, 2-D, 3-D spatial locality
    - Hardware range clamping, type conversion, interpolation


NVIDIA Kepler GPU
3-12 GB DRAM Memory w/ ECC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
1536KB Level 2 Cache
Graphics Processor Cluster
SMX
SMX
Streaming Multiprocessor - SMX
64 KB Constant Cache
64 KB L1 Cache / Shared Memory
48 KB Tex + Read-only Data Cache
SP
DP
LDST
SFU
Tex Unit
16 × Execution block =
192 SP, 64 DP,
32 SFU, 32 LDST

GPU Thread Block Collective Operations

- Threads within the same thread block can communicate with each other in fast on-chip **shared memory**
- Once scheduled on an SM, **thread blocks run until completion**
- Because the order of thread block execution is arbitrary and blocks cannot be stopped, **they cannot communicate or synchronize with other thread blocks (*)**
- **(*) Atomic memory ops are an exception wrt/ communication**

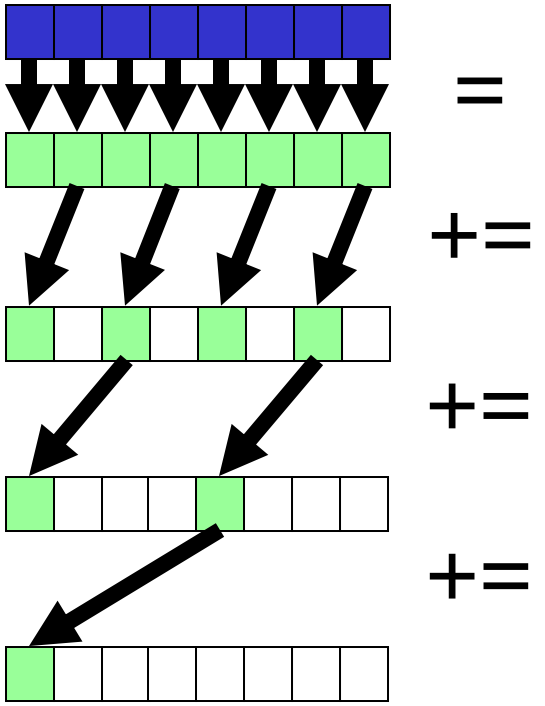
Thread blocks are multiplexed onto pool of GPU SMs…



Communication Between Threads

- Threads in a warp or a thread block can write/read shared memory, global memory
- **Barrier synchronizations**, and **memory fences** are used to ensure memory stores complete before peer(s) read…
- **Atomic ops** can enable limited communication between thread blocks

Shared Memory Parallel Reduction Example



Data Layout Issues and GPU Memory Systems

Array of Structures (AOS) vs. Structure of Arrays (SOA)

- AOS:

```
typedef struct {
  float x;
  float y;
  float z;
} myvec;
myvec aos[1024];
aos[threadIdx.x].x = 0;
aos[threadIdx.x].y = 0;
```

- SOA

```
typedef struct {
  float x[1024];
  float y[1024];
  float z[1024];
} myvecs;
myvecs soa;
soa.x[threadIdx.x] = 0;
soa.y[threadIdx.x] = 0;
```

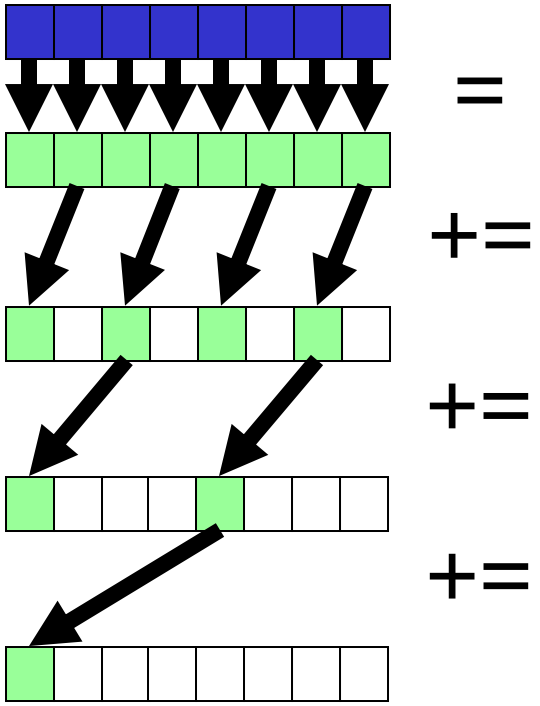
Use of Atomic Memory Ops

- Independent thread blocks can access shared counters, flags safely without deadlock when used properly
  - Allow a thread to inform peers to early-exit
  - Enable a thread block to determine that it is the last one running, and that it should do something special, e.g. a reduction of partial results from all thread blocks

Communication Between Threads in a Warp

- On GPUs since **Kepler**, neighboring threads in a warp can exchange data with each other using **shuffle instructions** between **registers**
- Shuffle outperforms shared memory, and leaves shared memory available for other data
- **CUDA 9: Cooperative Groups supercedes this approach…**

Intra-Warp Parallel Reduction with Shuffle, No Shared Memory Use



Avoid Output Conflicts, Conversion of Scatter to Gather

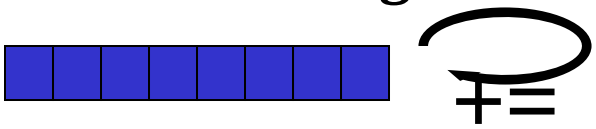
- Many CPU codes contain algorithms that "scatter" outputs to memory, to reduce arithmetic
- Scattered output can create bottlenecks for GPU performance due to bank conflicts
- On the GPU, it's often better to do **more arithmetic**, in exchange for a **regularized output pattern**, or to convert "scatter" algorithms to "gather" approaches

Avoid Output Conflicts: Privatization Schemes

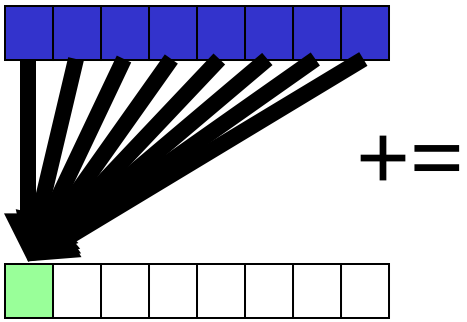
- ***Privatization***: use of private work areas for workers
  - Avoid/reduce the need for thread synchronization barriers
  - Avoid/reduce the need atomic increment/decrement operations during work, use **parallel reduction** at the end…
- By working in separate memory buffers, workers **avoid read/modify/write conflicts** of various kinds
- Huge GPU thread counts make it impractical to privatize data on a per-thread basis, so GPUs must use **coarser granularity: warps, thread-blocks**
- Use of the **on-chip shared memory** local to each SM can often be considered a form of privatization

Example: avoiding output conflicts when summing numbers among threads in a block

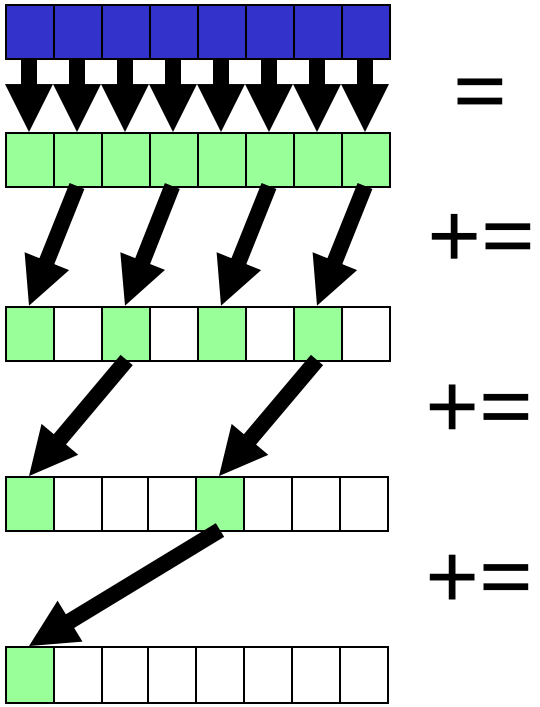
Accumulate sums in thread-local registers before doing any reduction among threads



N-way output conflict:
Correct results require **costly barrier synchronizations** or **atomic memory operations ON EVERY ADD** to prevent threads from overwriting each other…



Simple Parallel reduction:
no output conflicts,
Log2(N) barriers



Additional Reference Materials:

- https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/
- http://docs.nvidia.com/cuda/cuda-c-programming-guide
- The CUDA Handbook: A Comprehensive Guide to GPU Programming
  - https://www.amazon.com/CUDA-Handbook-Comprehensive-Guide-Programming/dp/0321809467
- Programming Massively Parallel Processors: A Hands-on Approach (Third Edition)
  - https://www.amazon.com/Programming-Massively-Parallel-Processors-Hands/dp/0128119861

Questions?

When to Use CUDA vs. OpenACC

Major Approaches For Programming Hybrid Architectures

- Use **drop-in libraries** in place of CPU-only libraries
  - **Little or no code development**
  - Examples: MAGMA, BLAS-variants, FFT libraries, etc.
  - **Speedups limited by Amdahl's Law** and overheads associated with data movement between CPUs and GPU accelerators
- Generate accelerator code as a variant of CPU source, e.g. using OpenMP and **OpenACC directives**, and similar
- Write **lower-level** accelerator-specific code, e.g. using **CUDA, OpenCL**, other approaches

Challenges Adapting Large Software Systems for State-of-the-Art Hardware Platforms

- Initial focus on key computational kernels eventually gives way to the need to optimize an **ocean of less critical routines**, due to observance of **Amdahl's Law**
- Even though these less critical routines might be easily ported to CUDA or similar, the **sheer number of routines often poses a challenge**
- Need a low-cost approach for **getting "some" speedup** out of these second-tier routines
- In many cases, it is completely **sufficient to achieve memory-bandwidth-bound GPU performance with an existing algorithm**

Amdahl's Law and Role of Directives

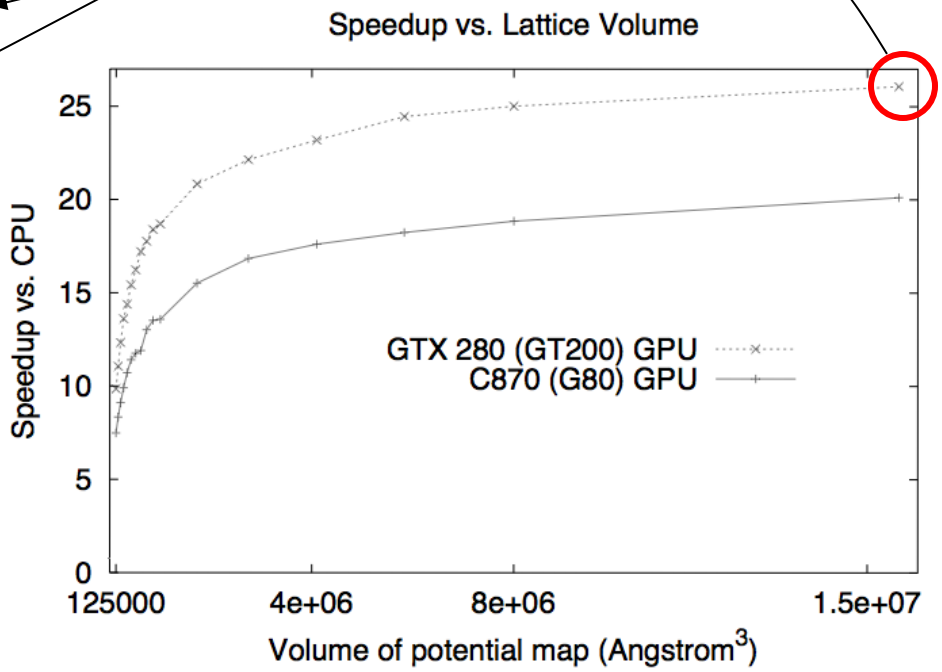
- Initial partitioning of algorithm(s) between host CPUs and accelerators is typically based on **initial performance balance point**
- **Time passes and accelerators get MUCH faster…**
- Formerly harmless CPU code ends up limiting overall performance!
- Need to address bottlenecks in increasing fraction of code
- **Directives** provide **low cost, low burden**, approach to **improve incrementally** vs. status quo
- **Directives are complementary to lower level approaches** such as CPU intrinsics, CUDA, OpenCL, and they all need to coexist and interoperate very gracefully alongside each other

Multilevel Summation on the GPU: An Amdahl's Law Example From Our Previous Work

Accelerate **short-range cutoff** and **lattice cutoff** parts

Performance profile for 0.5 Å map of potential for 1.5 M atoms. Hardware platform is Intel QX6700 CPU and NVIDIA GTX 280.

Computational steps	CPU (s)	w/ GPU (s)	Speedup
Short-range cutoff	480.07	14.87	32.3
Long-range anterpolation	0.18		
restriction	0.16		
lattice cutoff	49.47	1.36	36.4
prolongation	0.17		
interpolation	3.47		
Total	533.52	20.21	26.4



Multilevel summation of electrostatic potentials using graphics processing units. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

How Do Directives Fit In?

- Single code base is typically maintained
- Almost “deceptively” simple to use
- Easy route for **incremental, “gradual buy in”**
- **Rapid development cycle**, but success often follows minor refactoring and/or changes to data structure layout
- Higher abstraction level than other techniques for programming accelerators
- In many cases, **performance can be “good enough” due to memory-bandwidth limits**, or based on return on developer time or some other metric

Why Not Use Directives Exclusively?

- Some projects do…but:
  - Back-end runtimes for compiler directives sometimes have unexpected extra overheads that could be a showstopper in critical algorithm steps
  - High abstraction level may mean lack of access to hardware features exposed only via CUDA or other lower level APIs
  - Fortunately, **interoperability APIs** enable directive-based approaches to be used side-by-side with hand-coded kernels, libraries, etc.
  - Presently, sometimes-important capabilities like **JIT compilation of runtime-generated kernels** only exist within lower level APIs such as CUDA and OpenCL

What Do Existing Accelerated Applications Look Like?

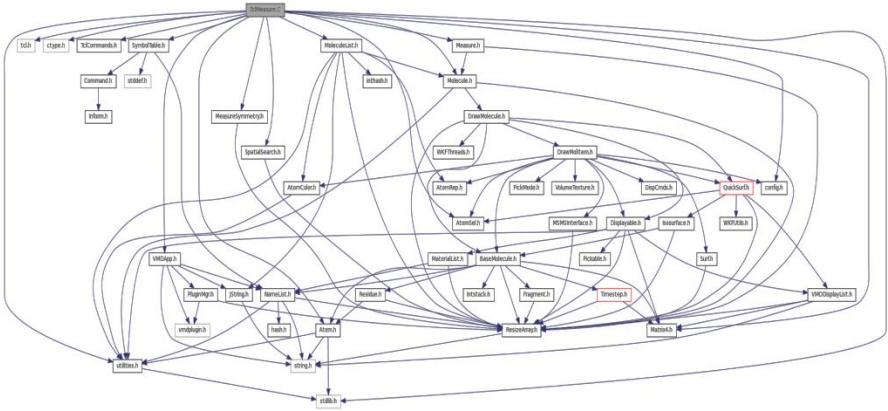
I'll provide examples from digging into modern versions of VMD that have already incorporated acceleration in a deep way.

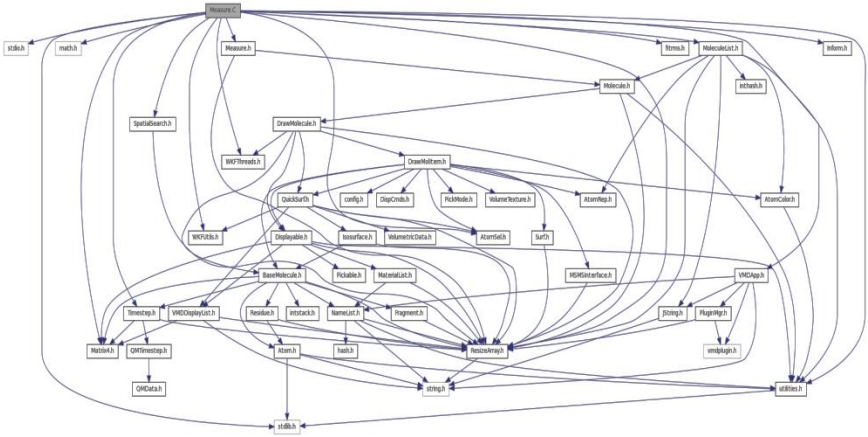
Questions:

- **How much code needs to be "fast", or "faster"**
- What fraction runs on accelerator now?
- Using directives, how much more coverage can be achieved, and with what speedup?
- Do I lose access to any points of execution or resource control that are critical for the application's performance?

Example of VMD Module Connectivity

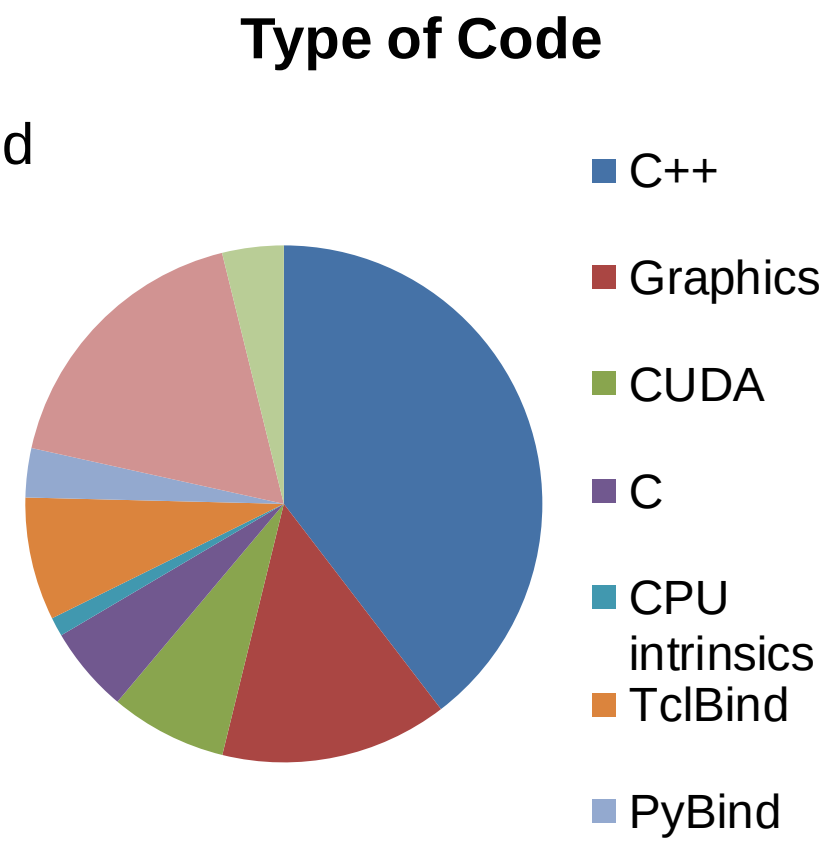
- Early progress focused acceleration efforts on handful of high level analysis routines that were the most computationally demanding
- Future hardware requires **pervasive acceleration**
- Top image shows script interface links to top level analytical routines
- Bottom image shows links among subset of data analytics algorithms to **leaf-node functions**





VMD Software Decomposition

- Computational code is 50% of VMD core
- Hand-written accelerator + vectorized code (CUDA + CPU intrinsics) represents only **14% of core computational code**
  - **20,000 lines of CUDA**
  - **3,100 lines of intrinics**
- Percent coverage of leaf-node analytical functions is lower yet
- Need to evolve VMD toward high coverage of performance-critical analysis code with fine-grained parallelism on accelerators and vectorization



Questions?