The CUDA Programming Model and Memory Hierarchy

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/Research/gpu/

Workshop on Accelerated High-Performance Computing in Computational Sciences (SMR 2760),

International Centre for Theoretical Physics (ICTP),

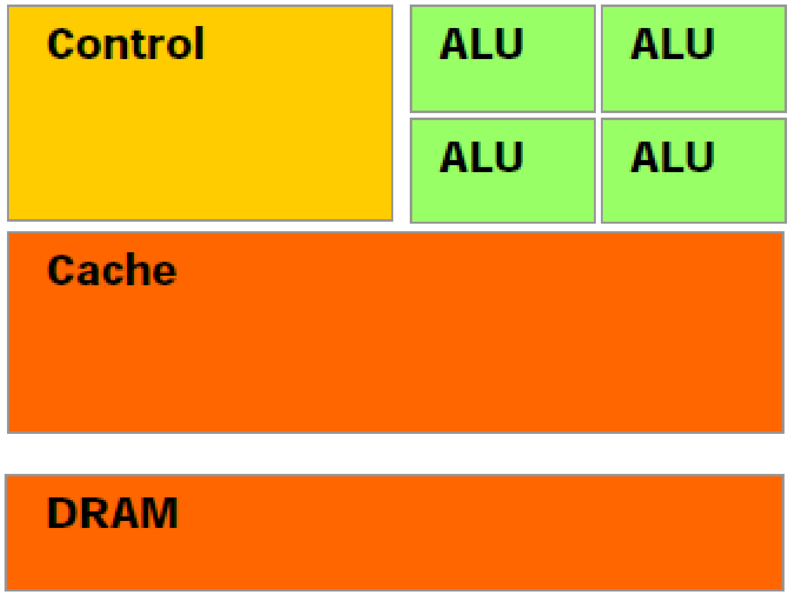
Trieste, Italy, May 27, 2015

CUDA Work Abstraction

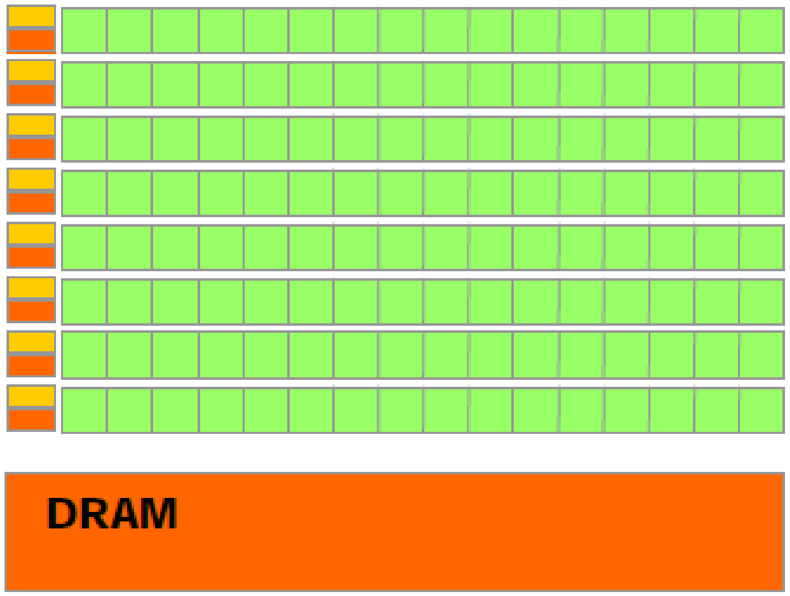
- Work is expressed as a multidimensional array of independent work items called **"threads"** – not the same thing as a CPU thread
- CUDA Kernels can be thought of as telling a GPU to compute **all iterations** of a set of nested loops **concurrently**
- Threads are dynamically scheduled onto hardware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

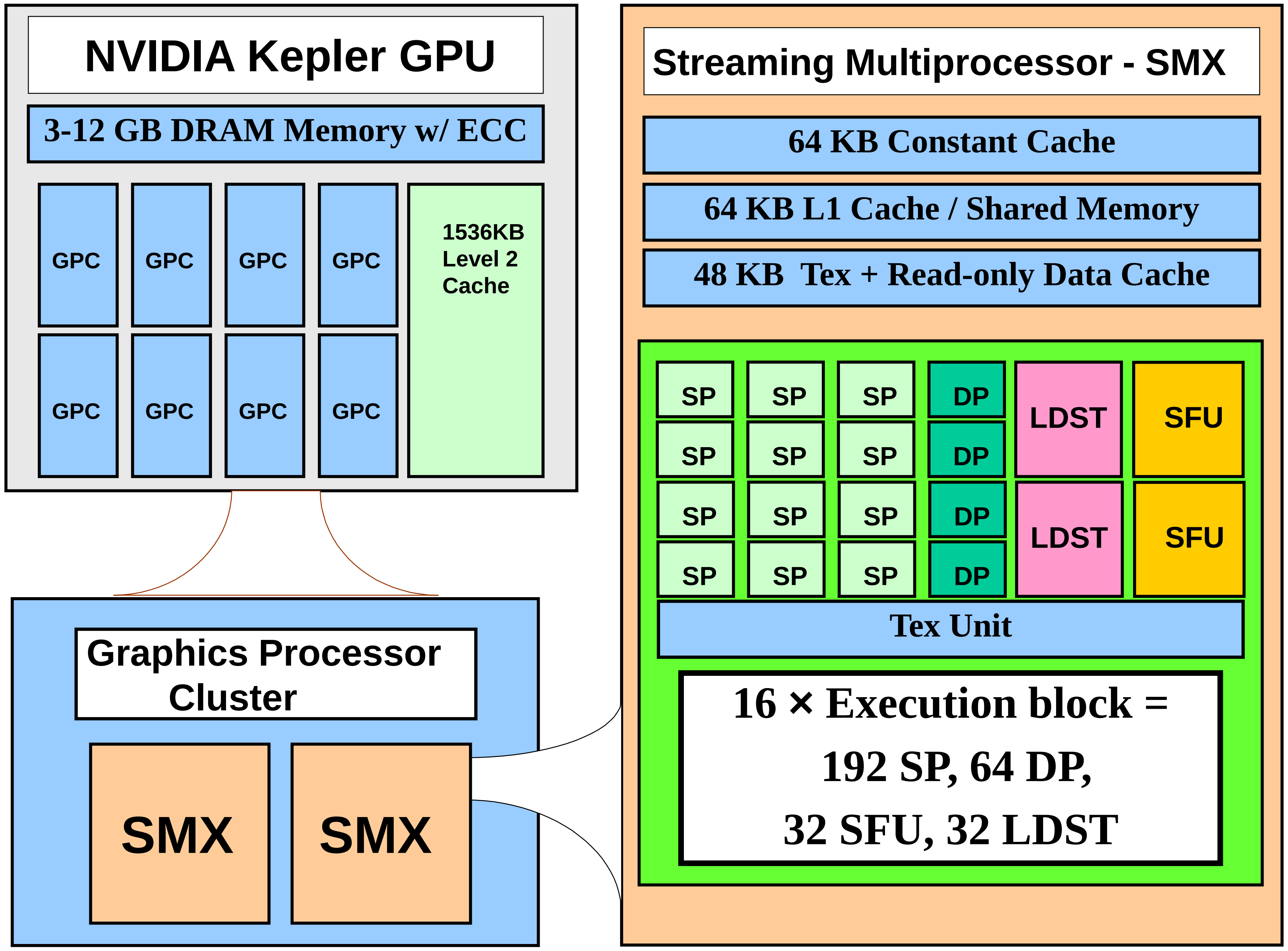
Comparison of CPU and GPU Hardware Architecture

CPU: Cache heavy, focused on individual thread performance



GPU: ALU heavy, massively parallel, throughput oriented




NVIDIA Kepler GPU
3-12 GB DRAM Memory w/ ECC
GPC
1536KB Level 2 Cache
Graphics Processor Cluster
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 "Occupancy"

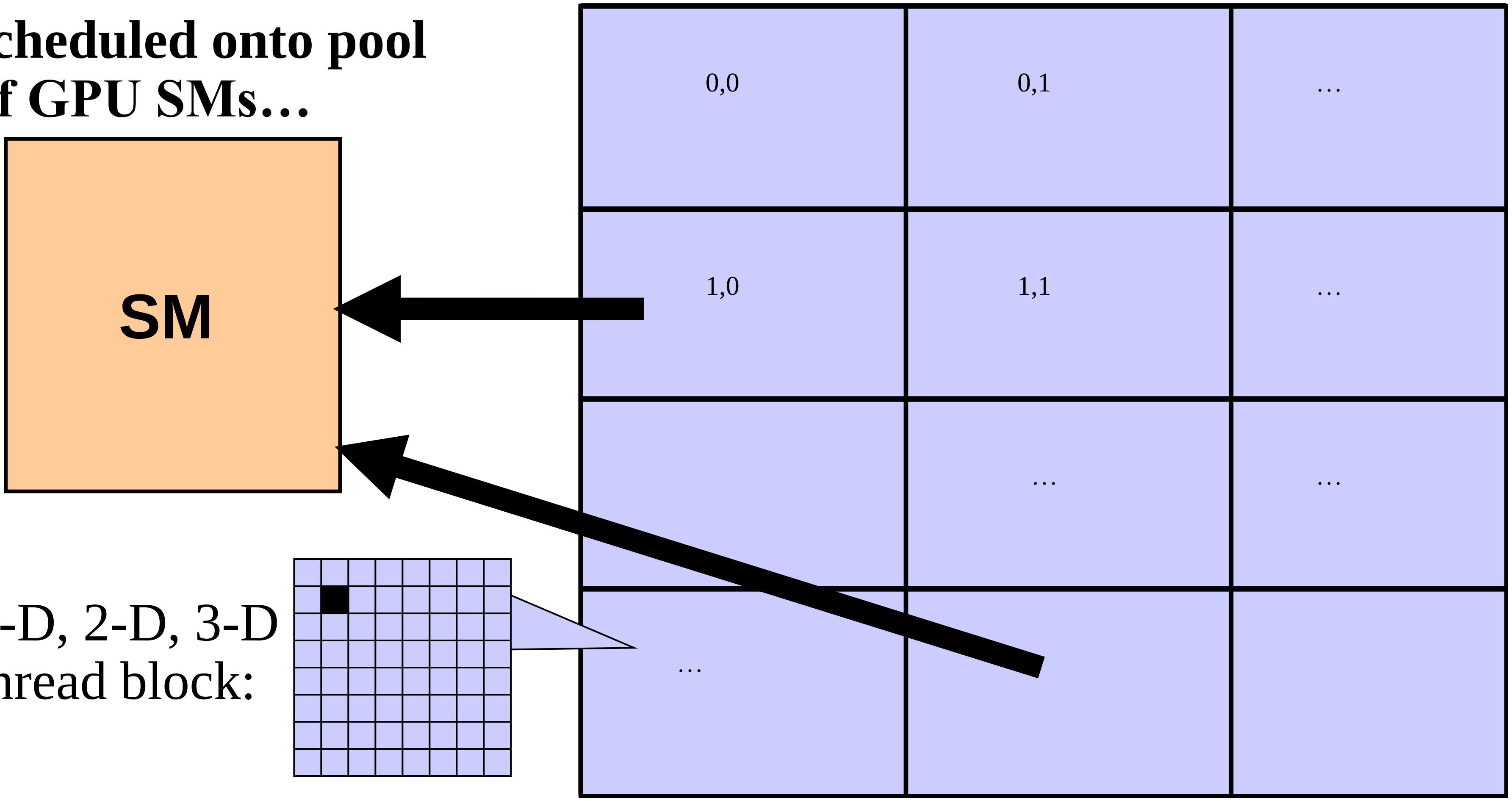
- GPU hardware designed to oversubscribe ALUs with lots of threads, thereby tolerating very long memory latencies without large on-chip caches
- Occupancy refers to the degree to which the GPUs warp scheduler is "full" of threads
- High occupancy often (but not always) provides greater latency hiding, which is usually (but not always) better for performance
- Sometimes it is possible to achieve good performance even with relatively low occupancy, via schemes that 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!!!

CUDA Work Abstractions: Grids, Thread Blocks, Threads

1-D, 2-D, or 3-D (SM >= 2.x)
Grid of thread blocks:

Thread blocks are scheduled onto pool of GPU SMs…

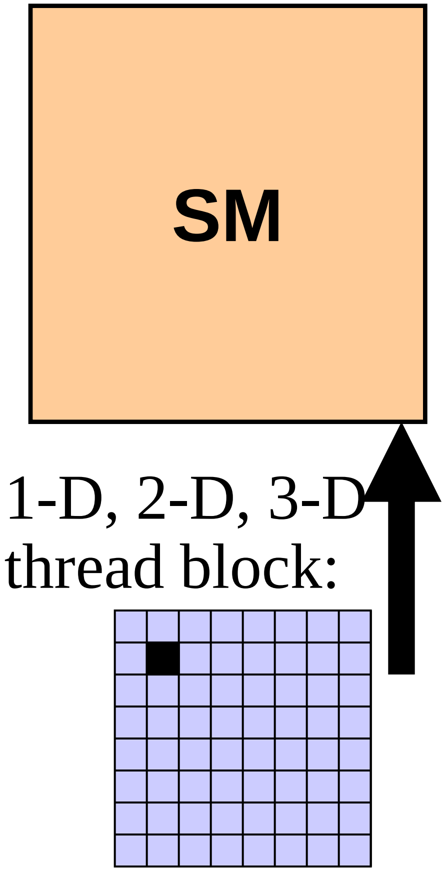
1-D, 2-D, 3-D thread block:



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 a thread has to be taken by all threads... (divergence is bad)**

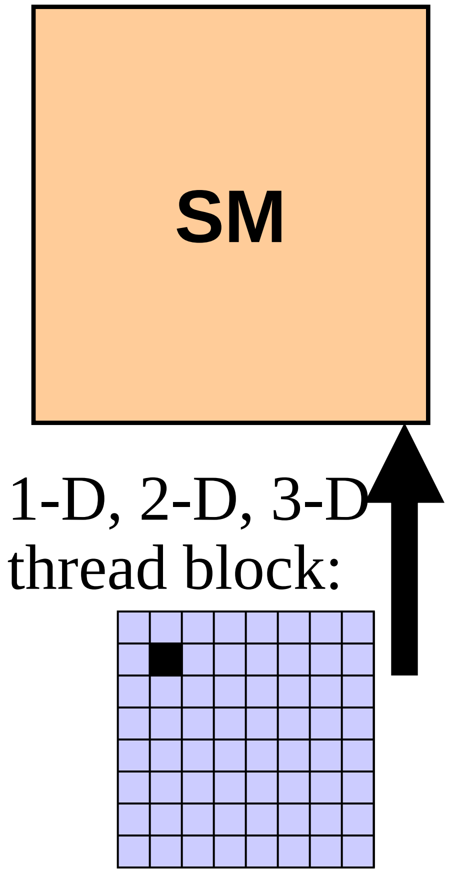
Thread blocks are multiplexed onto pool of GPU SMs…



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**
- Branch divergence issue not unique to GPUs, affects **all** SIMD hardware platforms…
- On GPUs, we get fast **hardware-based** implementation of predication/masking/etc…
- GPU 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 with all threads enabled...

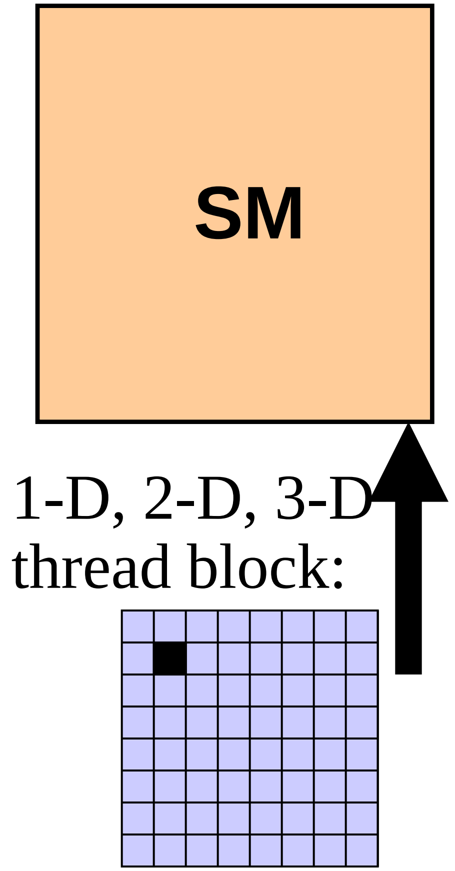
Thread blocks are multiplexed onto pool of GPU SMs…



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…

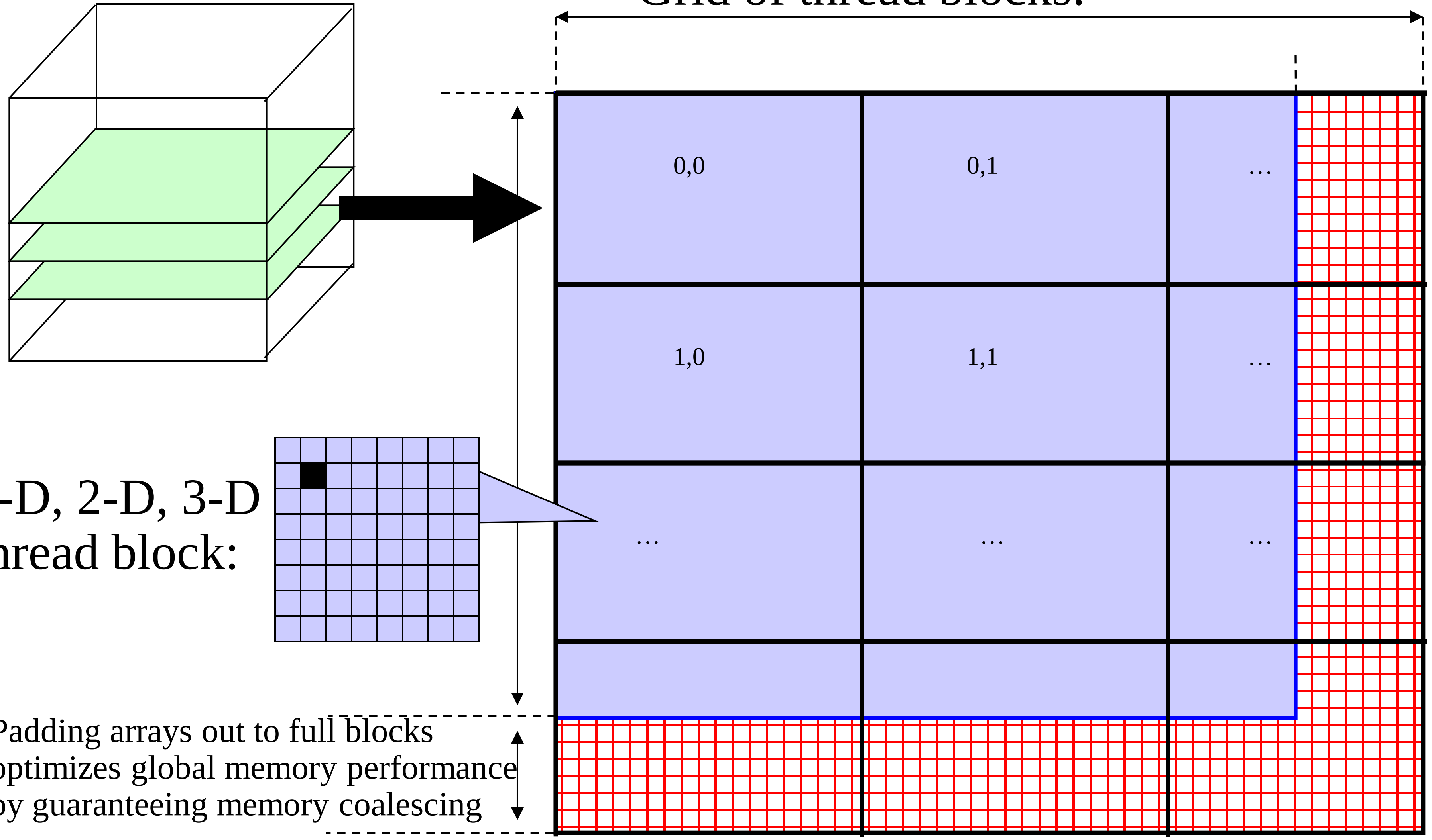


CUDA Grid/Block/Thread Decomposition

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

1-D, 2-D, or 3-D (SM >= 2.x)
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

Indexing Work

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

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, 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’…
```

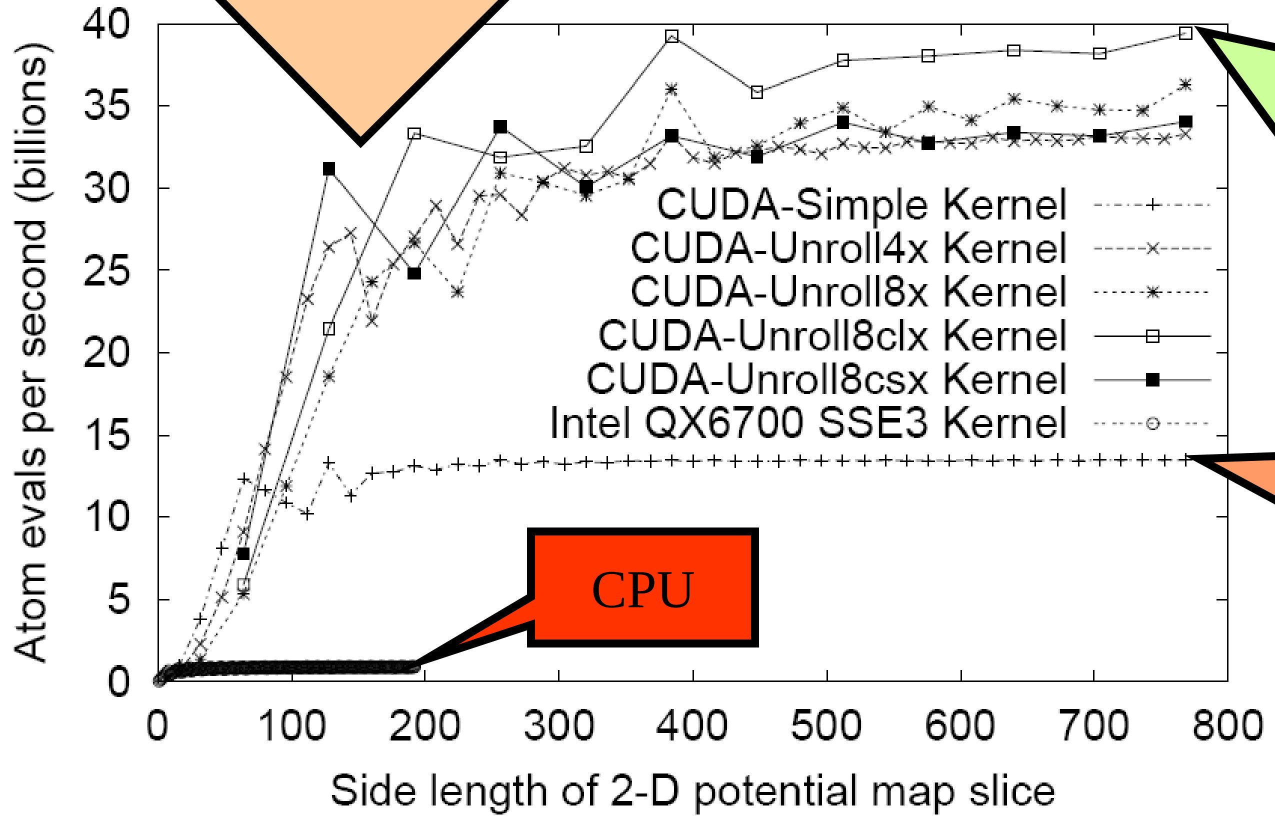
What if Work Size Isn't an Integer Multiple of the Thread Block Size?

- 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];
  }
}
```

Direct Coulomb Summation Performance

Number of thread blocks modulo number of SMs results in significant performance variation for small workloads



CUDA-Unroll8clx: fastest GPU kernel, 44x faster than CPU, 291 GFLOPS on GeForce 8800GTX

CUDA-Simple: 14.8x faster, 33% of fastest GPU kernel

GPU computing. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.

An Approach to Writing CUDA Kernels

- Find an algorithm that can expose **substantial parallelism**, we'll ultimately need tens of thousands of independent threads…
- Identify **appropriate** GPU memory or texture subsystems used to store data used by kernel
- Are there trade-offs that can be made to exchange computation for **more parallelism**?
  - Though counterintuitive, past successes resulted from this strategy
  - "Brute force" methods that expose significant parallelism do surprisingly well on GPUs
- Analyze the real-world use case for the problem and select a specialized kernel for the problem sizes that will be heavily used

Getting Performance From GPUs

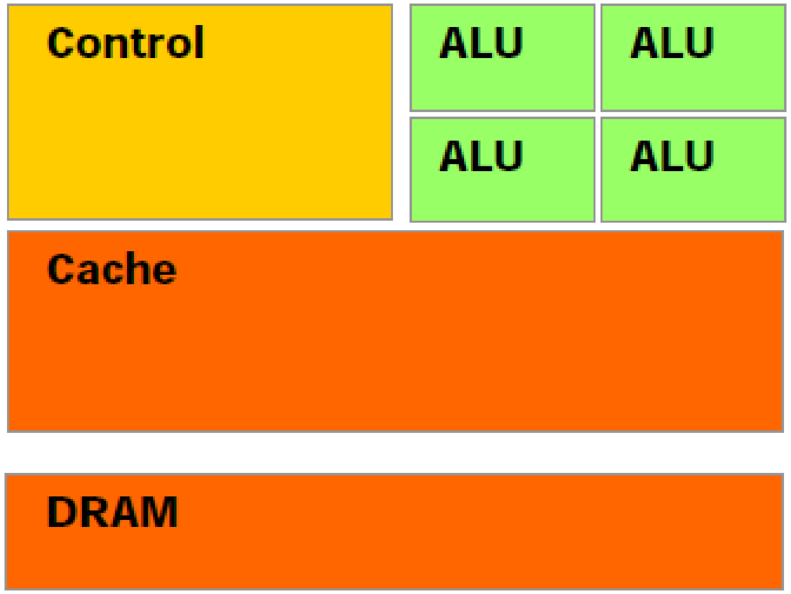
- Don’t worry (much) about counting arithmetic operations…at least until you have nothing else left to do
- GPUs provide tremendous memory bandwidth, but even so, **memory bandwidth often ends up being the performance limiter**
- Keep/reuse data in **registers** as long as possible
- The main consideration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

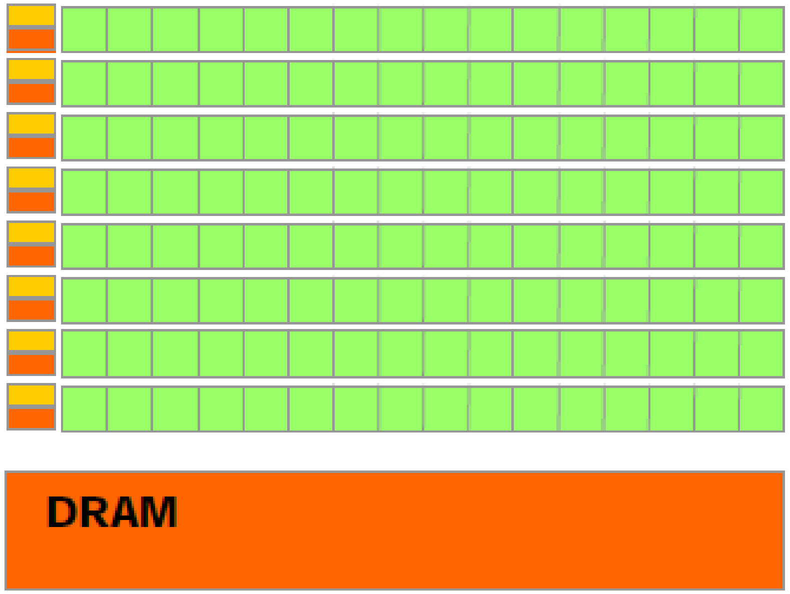
GPU Memory Systems

Comparison of CPU and GPU Hardware Architecture

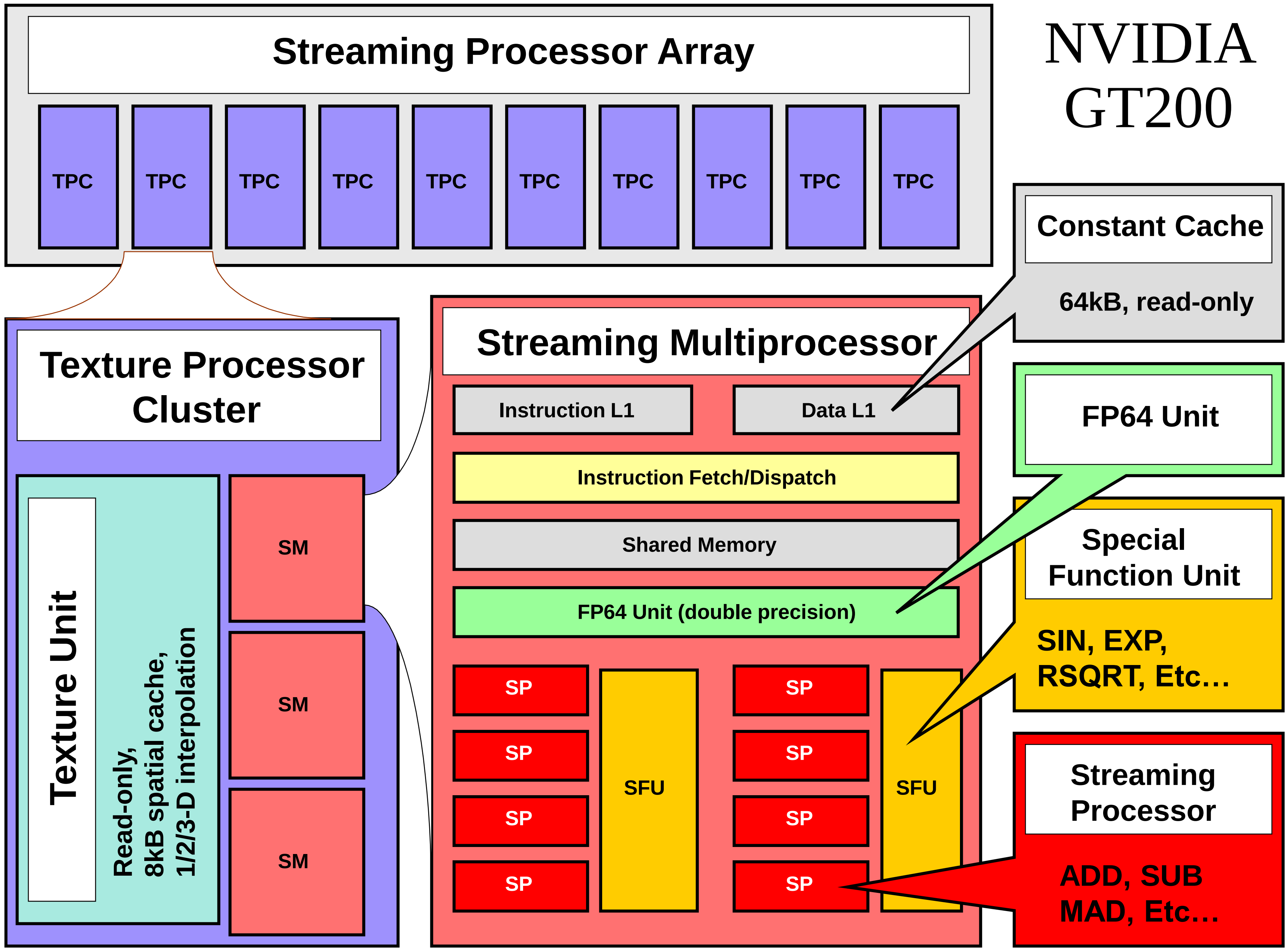
CPU: Cache heavy, focused on individual thread performance

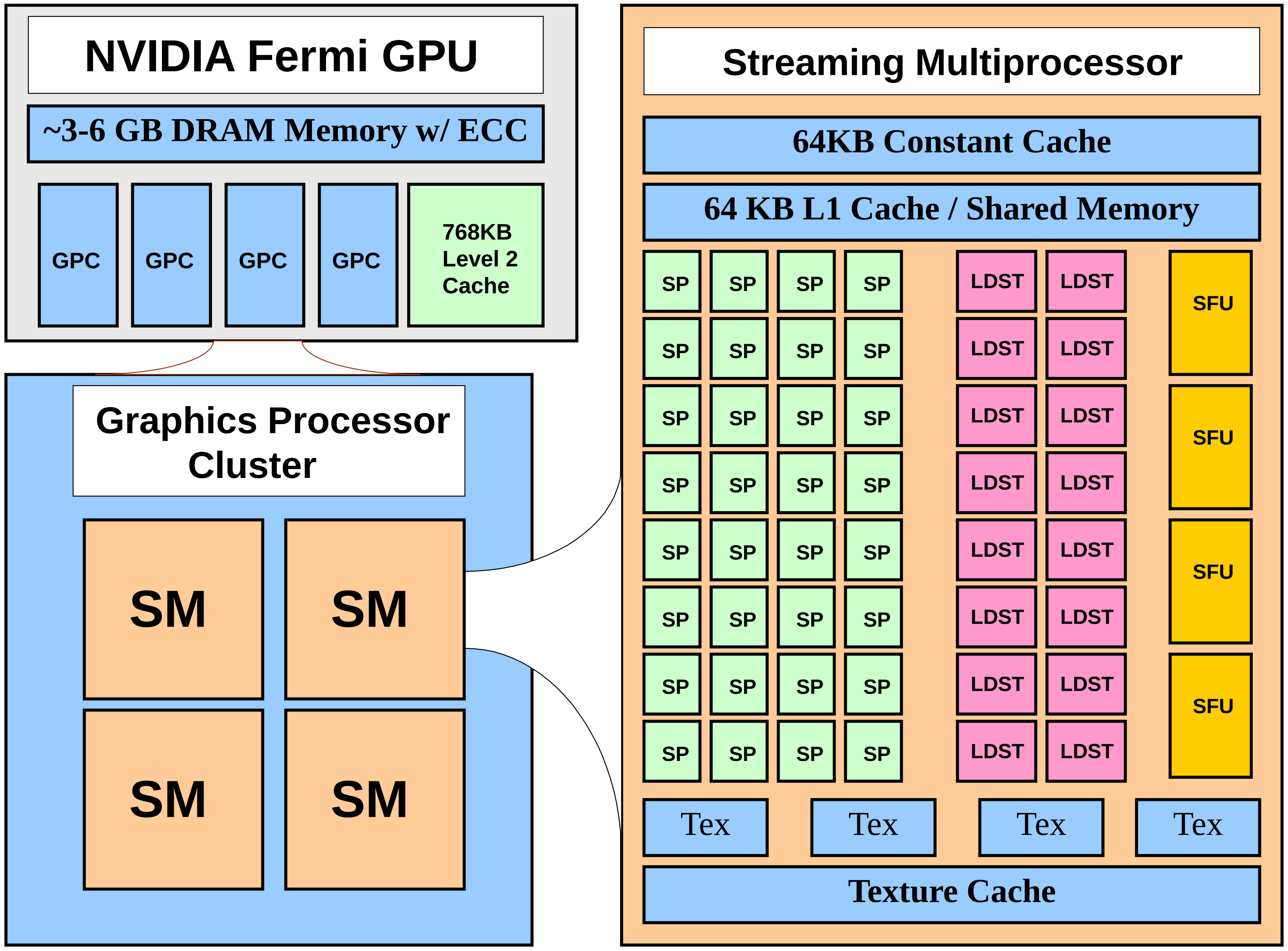
GPU: ALU heavy, massively parallel, throughput oriented

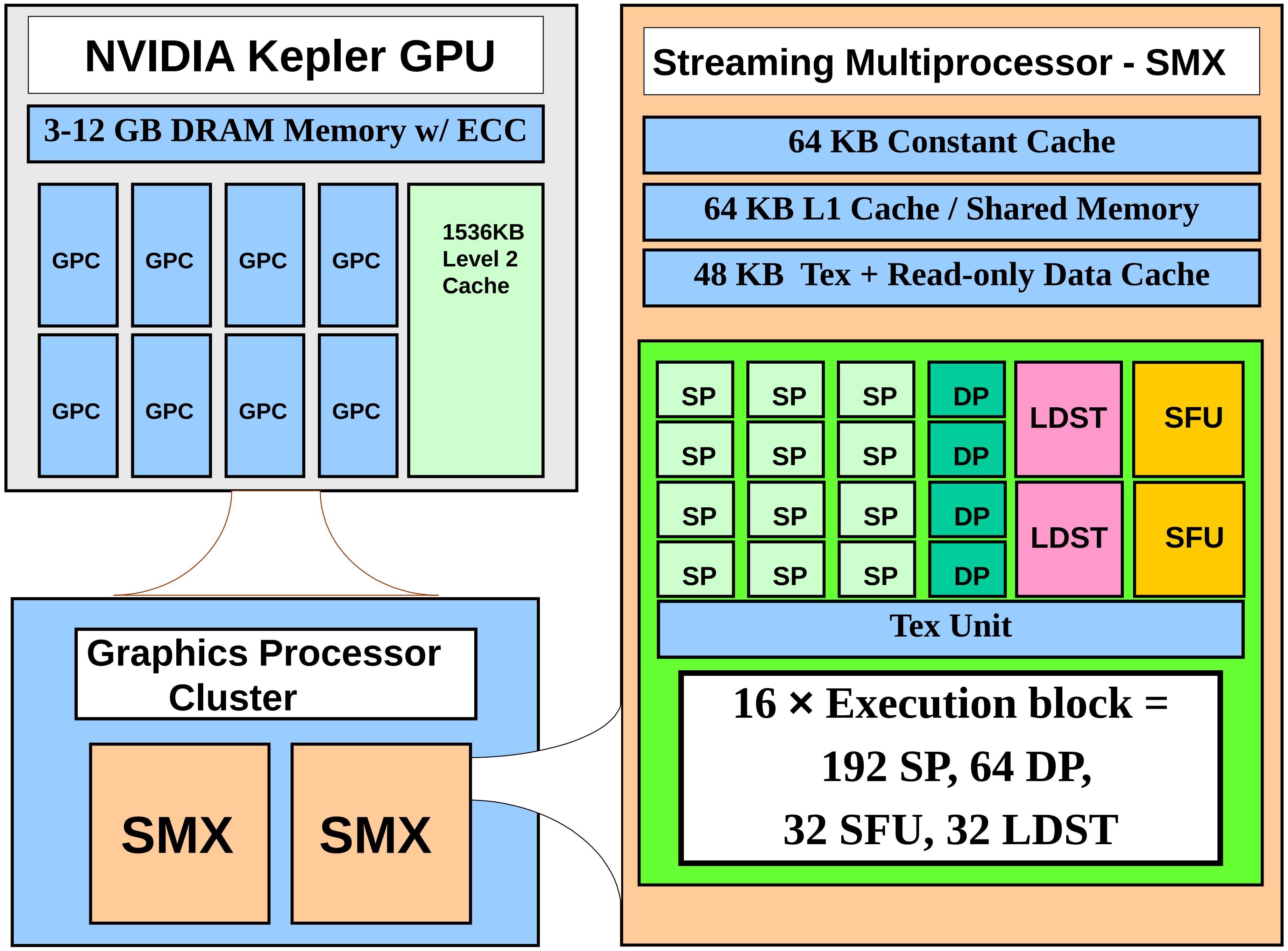




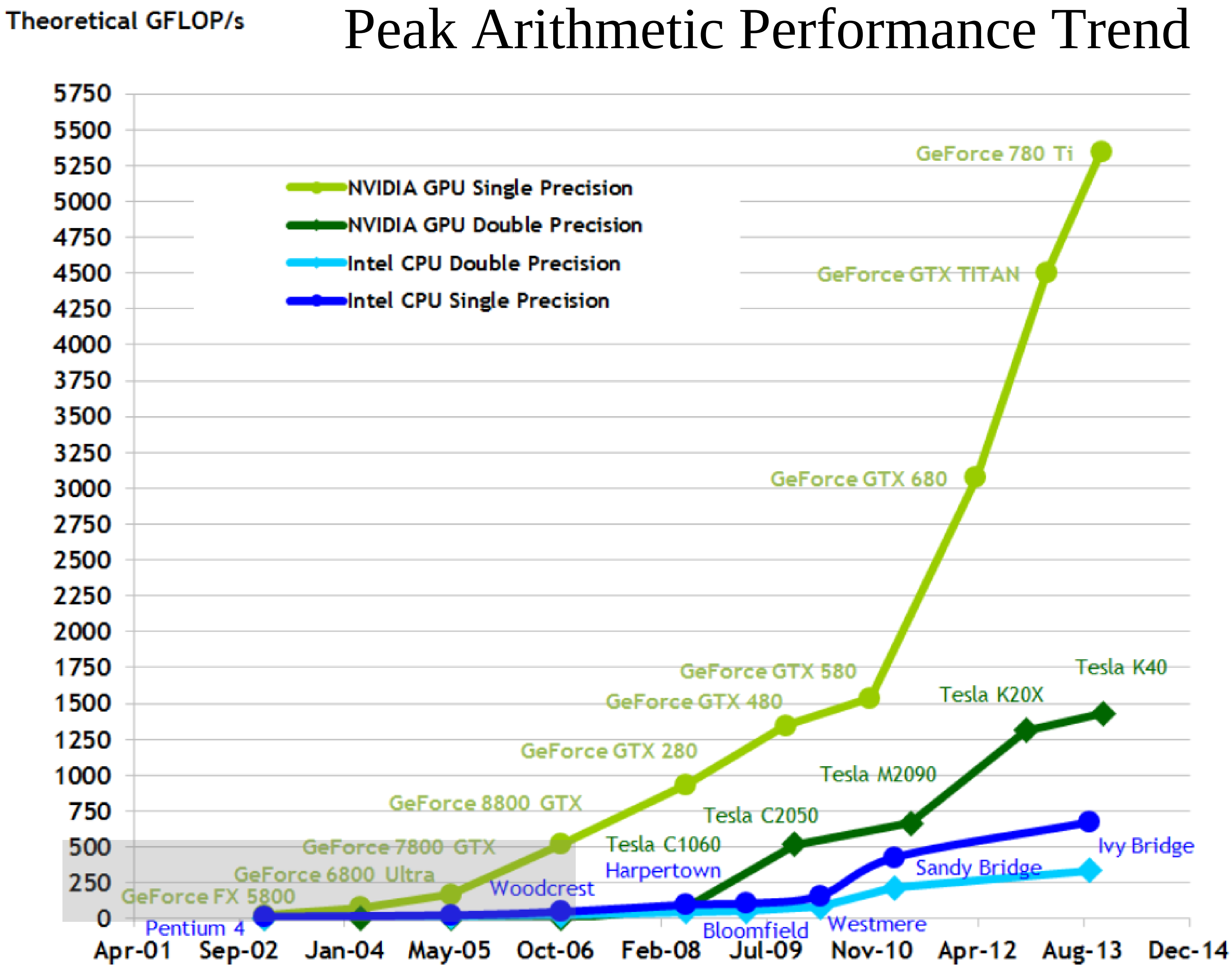
NVIDIA GT200



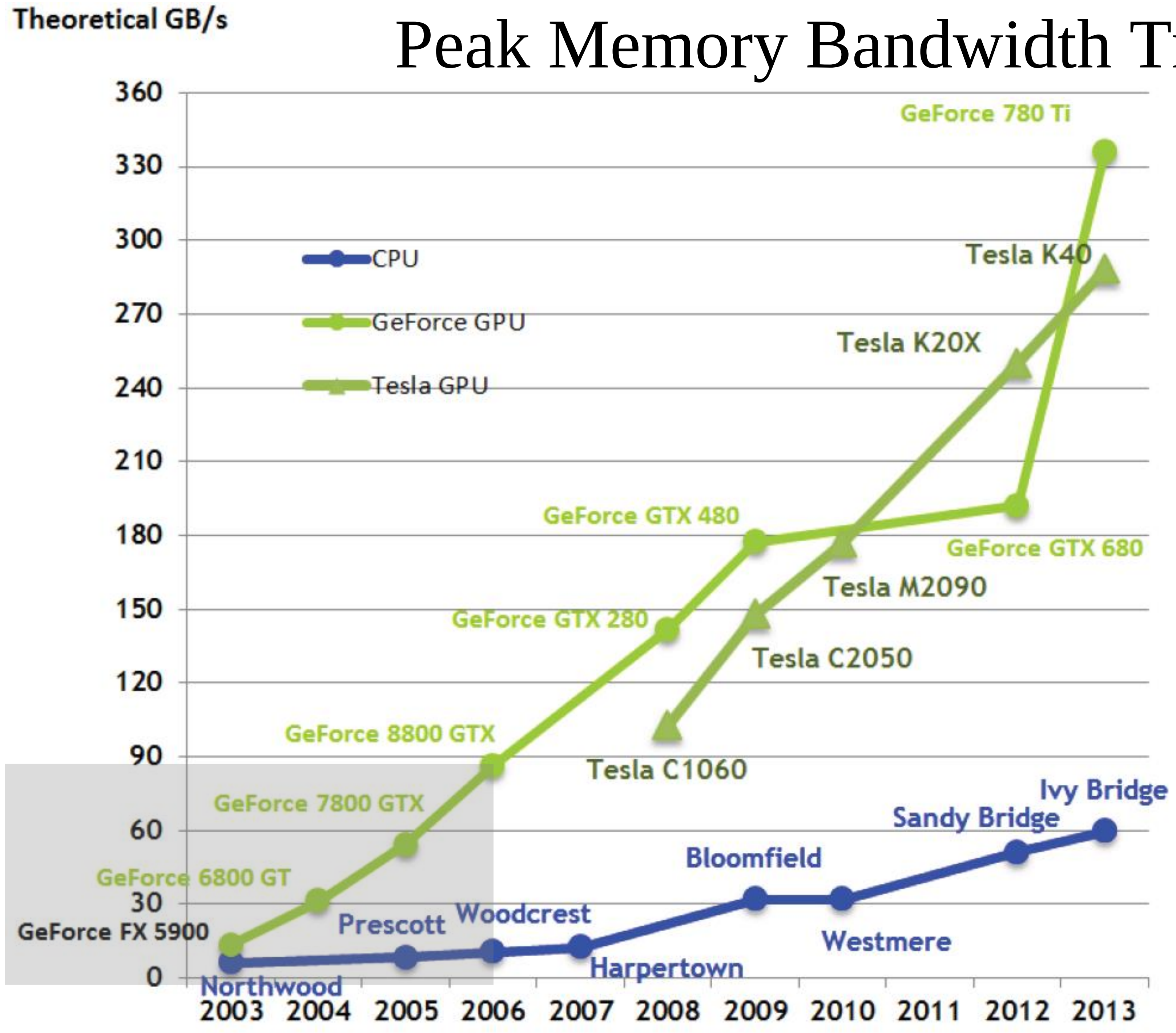

NVIDIA Fermi GPU
~3-6 GB DRAM Memory w/ ECC
GPC
GPC
GPC
GPC
768KB Level 2 Cache
Graphics Processor Cluster
SM
SM
SM
SM
Streaming Multiprocessor
64KB Constant Cache
64 KB L1 Cache / Shared Memory
SP
LDST
SFU
Tex
Tex
Tex
Tex
Texture Cache


NVIDIA Kepler GPU
3-12 GB DRAM Memory w/ ECC
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

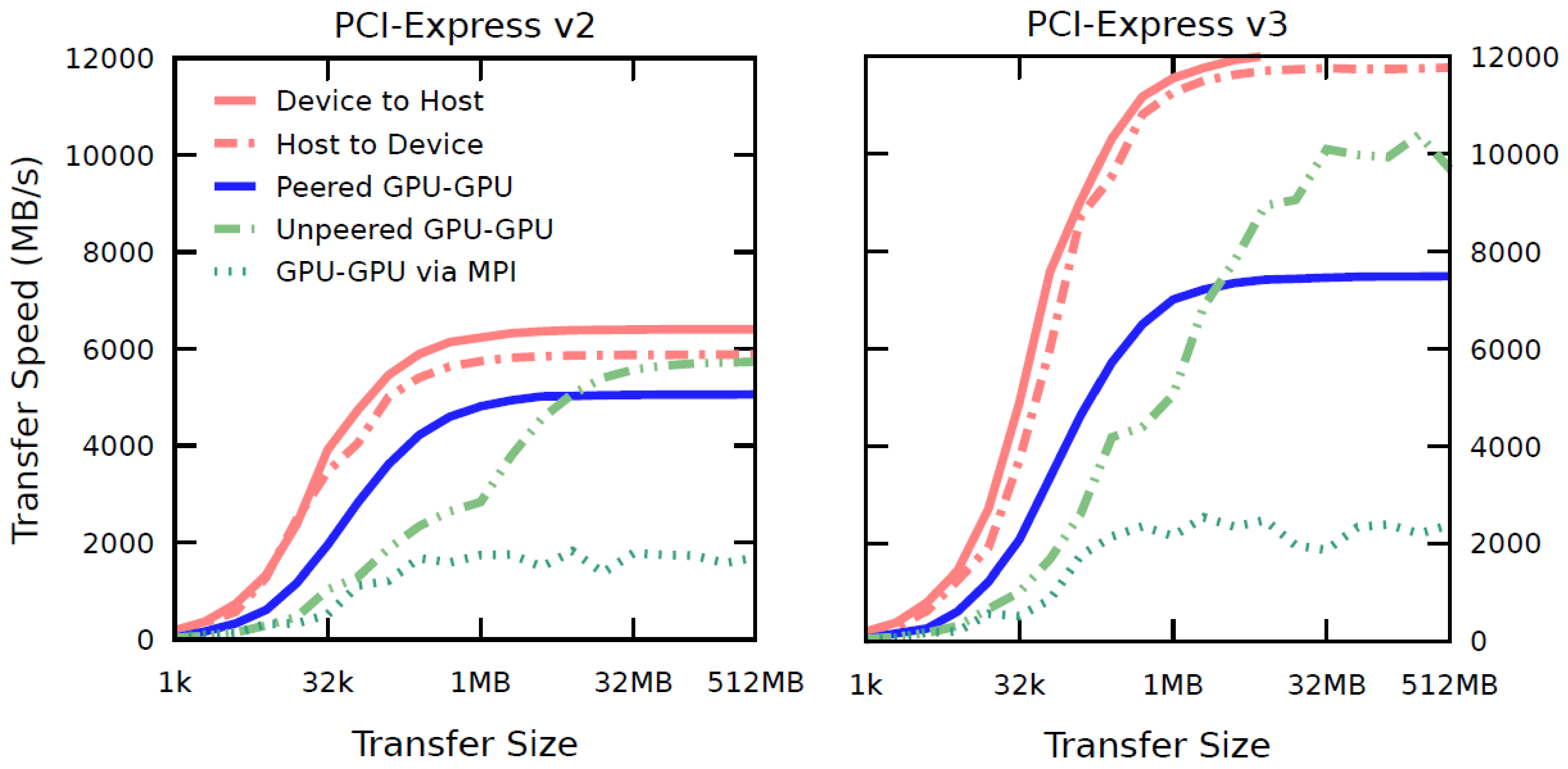
Peak Arithmetic Performance Trend



Peak Memory Bandwidth Trend



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, 2014. **(In press)**
http://dx.doi.org/10.1016/j.parco.2014.03.009

GPU On-Board Global Memory

- GPU arithmetic rates dwarf memory bandwidth
- For Kepler K40 hardware:
  - ~4.3 SP 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**

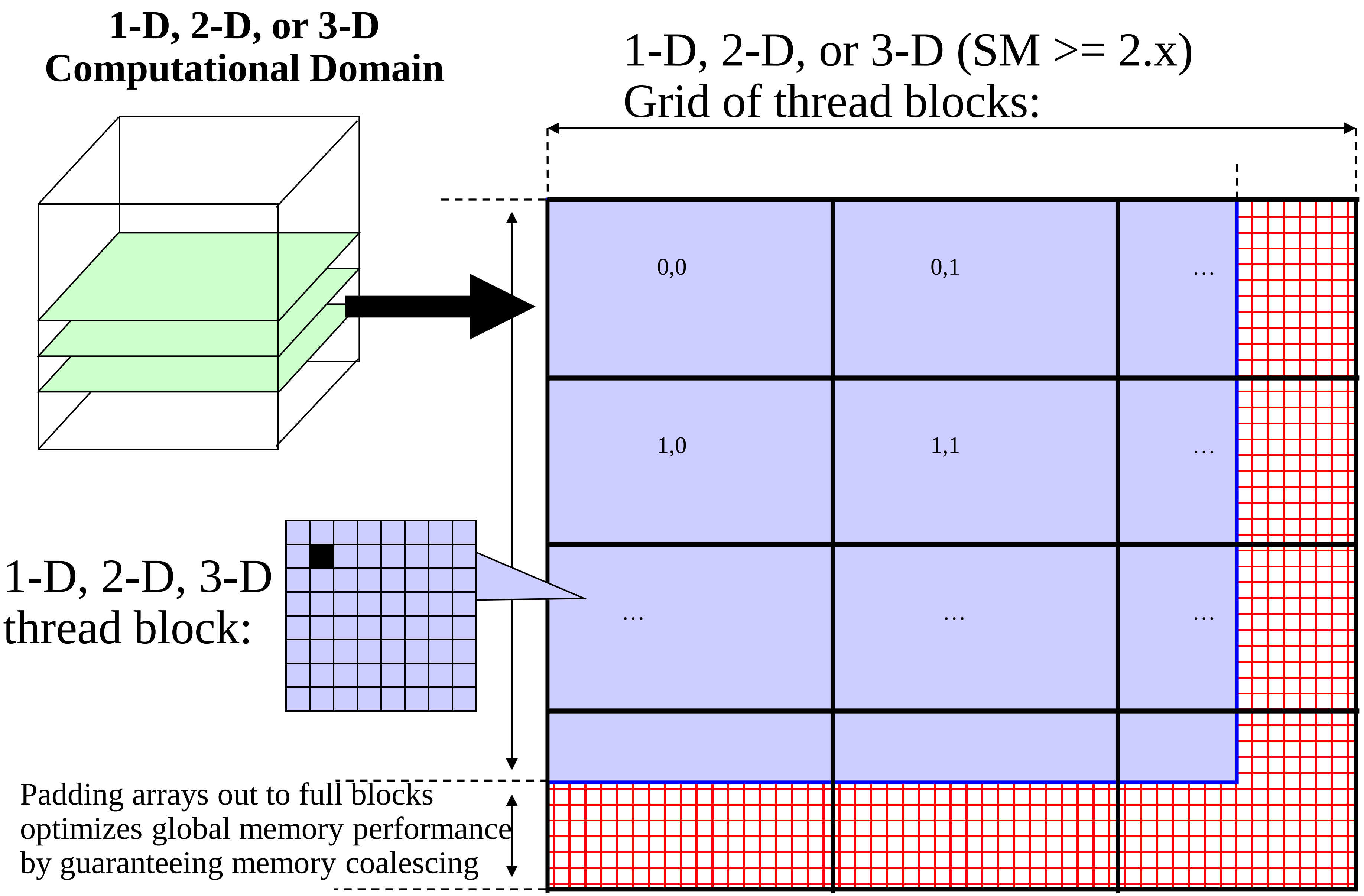
Memory Coalescing

- Oversimplified explanation:
  - Threads in a warp perform a read/write operation that can be serviced in a single hardware transaction
  - Rules vary slightly between hardware generations, but 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
  - Writes are similar, but multiple writes to the same location yields undefined results

Using the CPU to Optimize GPU Performance

- GPU performs best when the work evenly divides into the number of threads/processing units
- Optimization strategy:
  - Use the CPU to ***"regularize"*** the GPU workload
  - 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 highly occupied, attaining a high fraction of peak performance

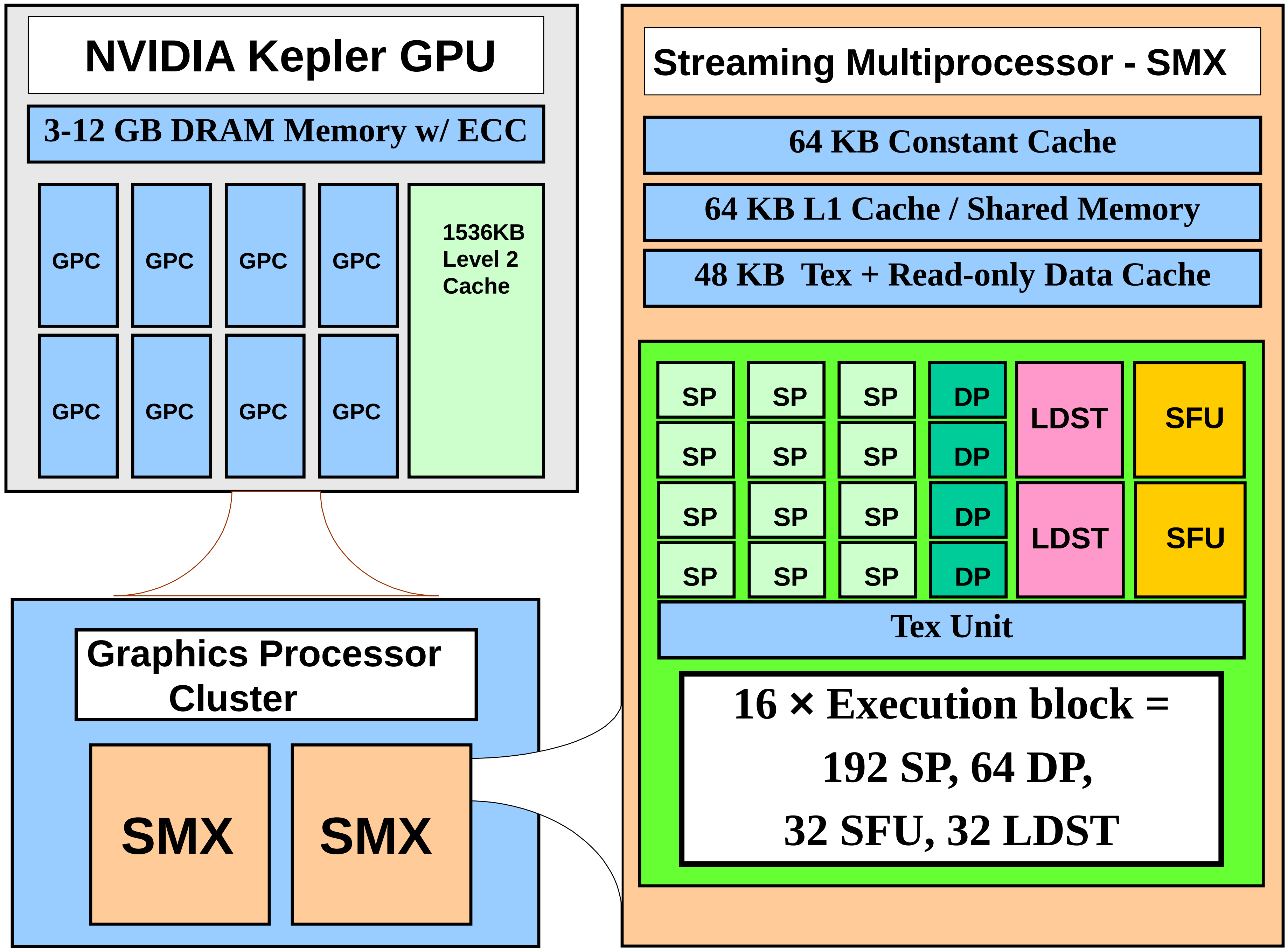
CUDA Grid/Block/Thread Decomposition



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

GPU On-Chip Memory Systems

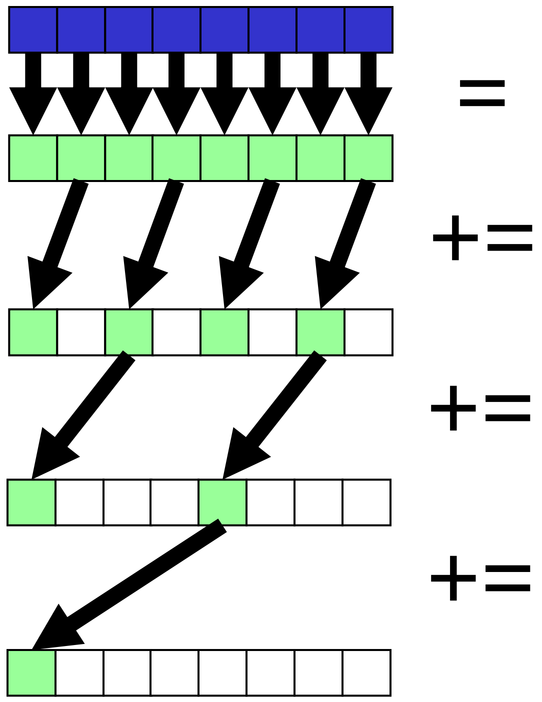
- GPU arithmetic rates dwarf global memory bandwidth
- GPUs include multiple fast **on-chip** memories to help **narrow the gap**:
  - **Registers**
  - Constant memory (64KB)
  - **Shared memory (48KB / 16KB)**
  - Read-only data cache / Texture cache (~48KB)
    - Hardware-assisted 1-D, 2-D, 3-D locality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


NVIDIA Kepler GPU
3-12 GB DRAM Memory w/ ECC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
GPC
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 SP SP DP
SP SP SP DP
SP SP SP DP
SP SP SP DP
LDST
SFU
LDST
SFU
Tex Unit
16 × Execution block = 192 SP, 64 DP, 32 SFU, 32 LDST

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



Example use of shared mem

```
__device__ float sumabsdiff_sumreduction(int tid, int totaltb,  float *sumabsdiffs_s, float *sumabsdiffs) {
 float sumabsdifftotal = 0.0f;
 if (tid < warpSize) {  // do the final reduction within a single warp only….
   for (int i=tid; i<totaltb; i+=warpSize) {
     sumabsdifftotal += sumabsdiffs[i];
   }
   sumabsdiffs_s[tid] = sumabsdifftotal;  // write to shared memory
 }
 __syncthreads(); // all threads must hit syncthreads call...
 // perform intra-warp parallel reduction...general loop version of parallel sum-reduction
 for (int s=warpSize>>1; s>0; s>>=1) {
   if (tid < s) {
     sumabsdiffs_s[tid] += sumabsdiffs_s[tid + s];
   }
   __syncthreads(); // all threads must hit syncthreads call...
 }
 return sumabsdiffs_s[0];
}
```

Avoiding Shared Memory Bank Conflicts: 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

Example use of atomic counters

```
__device__ unsigned int tbcatomic[3] = {0, 0, 0};  // GLOBAL ATOMIC COUNTER
VARIABLE
__device__ void reset_atomic_counter(unsigned int *counter) {
    counter[0] = 0;
    __threadfence();
}


…….
 #if __CUDA_ARCH__ >= 200
  // setup shared variable
  __shared__ bool isLastBlockDone;
  if (tid == 0)
    isLastBlockDone = 0;
  __syncthreads();
#endif
…….
```

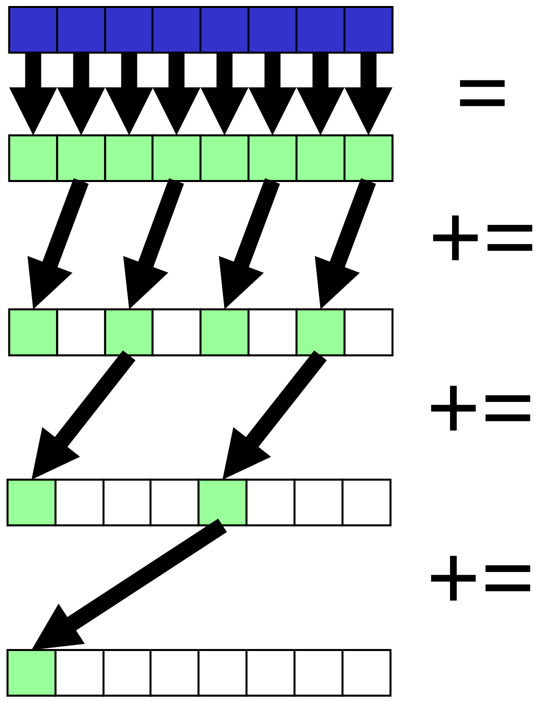
Example use of atomic counters

```
#if __CUDA_ARCH__ >= 200    // only compute capability 2.0 or greater have atomic ops
  if (tid == 0) {   // check if we are the last thread block to finish and finalize results
      unsigned int bid = blockIdx.z * gridDim.x * gridDim.y + blockIdx.y * gridDim.x + blockIdx.x;
      sumabsdiff[bid] = sumabsdiff_s[0];
      __threadfence();
      unsigned int value = atomicInc(&tbcatomic[0], totaltb);
      isLastBlockDone = (value == (totaltb - 1));
  }
  __syncthreads();
  if (isLastBlockDone) {
      float totalsumabsdiff = sumabsdiff_sumreduction(tid, totaltb, sumabsdiff_s, sumabsdiff);
      if (tid == 0)
          sumabsdiff[totaltb] = totalsumabsdiff;
      reset_atomic_counter(&tbcatomic[0]);
  }
#endif
```

Communication Between Threads in a Warp

- On the recent **Kepler/Maxwell** GPUs, neighboring threads in a warp can exchange data with each other using **shuffle instructions**
- Shuffle outperforms shared memory, and leaves shared memory available for other data

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



Avoid Output Conflicts, Conversion of Scatter to Gather

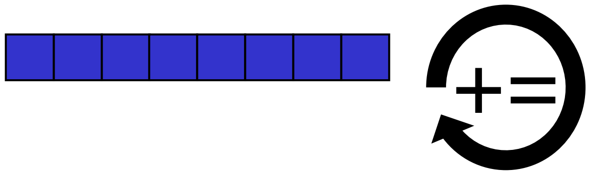
- Many CPU codes contain algorithms that “scatter” outputs to memory, to reduce arithmetic
- Scattered output can create bottlenecks for GPU performance due to bank conflicts
- On the GPU, it’s often better to do **more arithmetic**, in exchange for a **regularized output pattern**, or to convert “scatter”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

Avoid Output Conflicts: Privatization Schemes

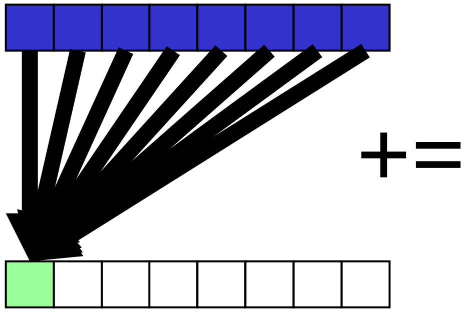
- ***Privatization***: use of private work areas for workers
  - Avoid/reduce the need for thread synchronization barriers
  - Avoid/reduce the need atomic increment/decrement operations during work, use **parallel reduction** at the end…
- By working in separate memory buffers, workers **avoid read/modify/write 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

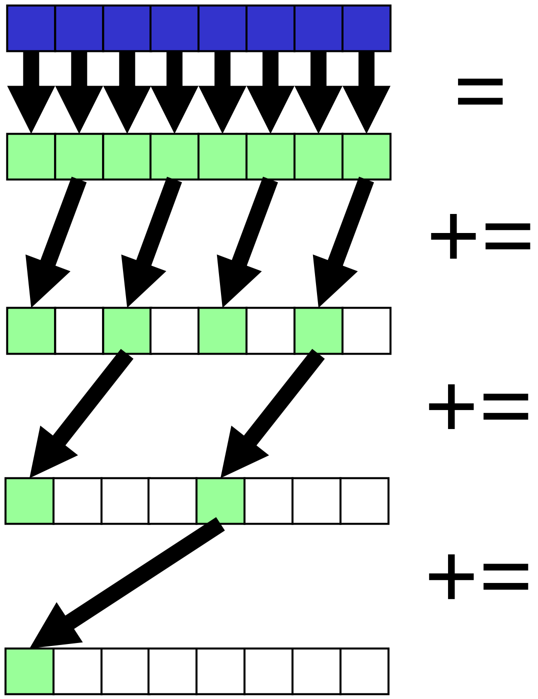
Accumulate sums in thread-local registers before doing any reduction among threads



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



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



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

Multi-GPU Cards Will Become More Common

- Peak memory bandwidth bound by area / perimeter of the GPU processor die, pin count, etc.
- Apps bound by memory bandwidth can be better served by multi-GPU systems, multi-GPU cards
- Multi-GPU cards will likely become more common as die-stacked memory and fast GPU-to-GPU links (e.g. announced NVLink) arrive

GeForce Titan Z

Tesla K10

GeForce GTX 690

Acknowledgements

- Theoretical and Computational Biophysics Group, University of Illinois at Urbana-Champaign
- NVIDIA CUDA Center of Excellence, University of Illinois at Urbana-Champaign
- NVIDIA CUDA team
- NCSA Blue Waters Team
- Funding:
  - NSF OCI 07-25070
  - NSF PRAC “The Computational Microscope”
  - NIH support: 9P41GM104601, 5R01GM098243-02

NIH BTRC for Macromolecular Modeling and Bioinformatics
1990-2017
Beckman Institute
University of Illinois at Urbana-Champaign

GPU Computing Publications

http://www.ks.uiuc.edu/Research/gpu/

- **Runtime and Architecture Support for Efficient Data Exchange in Multi-Accelerator Applications** Javier Cabezas, Isaac Gelado, John E. Stone, Nacho Navarro, David B. Kirk, and Wen-mei Hwu. IEEE Transactions on Parallel and Distributed Systems, 26(5):1405-1418, 2015.
- **Unlocking the Full Potential of the Cray XK7 Accelerator** Mark Klein and John E. Stone. Cray Users Group, Lugano Switzerland, May 2014.
- **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.
- **GPU-Accelerated Analysis and Visualization of Large Structures Solved by Molecular Dynamics Flexible Fitting** John E. Stone, Ryan McGreevy, Barry Isralewitz, and Klaus Schulten. Faraday Discussions, 169:265-283, 2014.
- **GPU-Accelerated Molecular Visualization on Petascale Supercomputing Platforms.** J. Stone, K. L. Vandivort, and K. Schulten. UltraVis'13: Proceedings of the 8th International Workshop on Ultrascale Visualization, pp. 6:1-6:8, 2013.
- **Early Experiences Scaling VMD Molecular Visualization and Analysis Jobs on Blue Waters.** J. E. Stone, B. Isralewitz, and K. Schulten. Extreme Scaling Workshop (XSW), pp. 43-50, 2013.
- **Lattice Microbes: High-performance stochastic simulation method for the reaction-diffusion master equation.** E. Roberts, J. E. Stone, and Z. Luthey-Schulten. J. Computational Chemistry 34 (3), 245-255, 2013.

GPU Computing Publications

http://www.ks.uiuc.edu/Research/gpu/

- **Fast Visualization of Gaussian Density Surfaces for Molecular Dynamics and Particle System Trajectories.** M. Krone, J. E. Stone, T. Ertl, and K. Schulten. *EuroVis Short Papers*, pp. 67-71, 2012.
- **Fast Analysis of Molecular Dynamics Trajectories with Graphics Processing Units – Radial Distribution Functions.** B. Levine, J. Stone, and A. Kohlmeyer. *J. Comp. Physics*, 230(9):3556-3569, 2011.
- **Immersive Out-of-Core Visualization of Large-Size and Long-Timescale Molecular Dynamics Trajectories.** J. Stone, K. Vandivort, and K. Schulten. G. Bebis et al. (Eds.): *7th International Symposium on Visual Computing (ISVC 2011)*, LNCS 6939, pp. 1-12, 2011.
- **Quantifying the Impact of GPUs on Performance and Energy Efficiency in HPC Clusters.** J. Enos, C. Steffen, J. Fullop, M. Showerman, G. Shi, K. Esler, V. Kindratenko, J. Stone, J Phillips. *International Conference on Green Computing*, pp. 317-324, 2010.
- **GPU-accelerated molecular modeling coming of age. J. Stone, D. Hardy, I. Ufimtsev, K. Schulten.** *J. Molecular Graphics and Modeling*, 29:116-125, 2010.
- **OpenCL: A Parallel Programming Standard for Heterogeneous Computing. J. Stone, D. Gohara, G. Shi.** *Computing in Science and Engineering*, 12(3):66-73, 2010.

GPU Computing Publications
http://www.ks.uiuc.edu/Research/gpu/

- **An Asymmetric Distributed Shared Memory Model for Heterogeneous Computing Systems**. I. Gelado, J. Stone, J. Cabezas, S. Patel, N. Navarro, W. Hwu. *ASPLOS ’10: Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 347-358, 2010.
- **GPU Clusters for High Performance Computing**. V. Kindratenko, J. Enos, G. Shi, M. Showerman, G. Arnold, J. Stone, J. Phillips, W. Hwu. *Workshop on Parallel Programming on Accelerator Clusters (PPAC)*, In Proceedings IEEE Cluster 2009, pp. 1-8, Aug. 2009.
- **Long time-scale simulations of in vivo diffusion using GPU hardware**. E. Roberts, J. Stone, L. Sepulveda, W. Hwu, Z. Luthey-Schulten. In *IPDPS’09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Computing*, pp. 1-8, 2009.
- **High Performance Computation and Interactive Display of Molecular Orbitals on GPUs and Multi-core CPUs**. J. Stone, J. Saam, D. Hardy, K. Vandivort, W. Hwu, K. Schulten, *2nd Workshop on General-Purpose Computation on Graphics Pricessing Units (GPGPU-2), ACM International Conference Proceeding Series*, volume 383, pp. 9-18, 2009.
- **Probing Biomolecular Machines with Graphics Processors**. J. Phillips, J. Stone. *Communications of the ACM*, 52(10):34-41, 2009.
- **Multilevel summation of electrostatic potentials using graphics processing units**. D. Hardy, J. Stone, K. Schulten. *J. Parallel Computing*, 35:164-177, 2009.

GPU Computing Publications
http://www.ks.uiuc.edu/Research/gpu/

- **Adapting a message-driven parallel application to GPU-accelerated clusters**. J. Phillips, J. Stone, K. Schulten. *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, IEEE Press, 2008.
- **GPU acceleration of cutoff pair potentials for molecular modeling applications**. C. Rodrigues, D. Hardy, J. Stone, K. Schulten, and W. Hwu. *Proceedings of the 2008 Conference On Computing Frontiers*, pp. 273-282, 2008.
- **GPU computing**. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, J. Phillips. *Proceedings of the IEEE*, 96:879-899, 2008.
- **Accelerating molecular modeling applications with graphics processors**. J. Stone, J. Phillips, P. Freddolino, D. Hardy, L. Trabuco, K. Schulten. *J. Comp. Chem.*, 28:2618-2640, 2007.
- **Continuous fluorescence microphotolysis and correlation spectroscopy**. A. Arkhipov, J. Hüve, M. Kahms, R. Peters, K. Schulten. *Biophysical Journal*, 93:4006-4017, 2007.