Lecture 16 – OpenMP Constructs

- **OpenMP's constructs fall in 5 categories:**
  - Parallel regions
  - Synchronization
  - Worksharing
  - Data environment

  } We discussed these in the last lecture.

  - Runtime functions/environment variables

Runtime Functions

- **Run time library can be used to control and query the parallel execution environment**
  - OMP_SET_NUM_THREADS()
  - OMP_GET_NUM_THREADS()
  - OMP_GET_THREAD_NUM()
  - OMP_GET_NUM_PROCS()
  - OMP_IN_PARALLEL(): *return true or false*
- **When calling these OMP functions from a C-code, you need to**

```
#include <omp.h>
```

  - On hpc1, omp.h is located at /share/apps/pgi-2015/linux86-64/15.4/include. You probably need to use the pgi compiler in the /share/apps/pgi-2015/linux86-64/15.4/bin directory.
  - You can point to the omp.h file by adding the compiler option

-I/share/apps/pgi-2015/linux86-64/15.4/include

Programming Errors

- **Shared memory parallel programming is a mixed bag**
  - It saves programmer from mapping data onto multiple processors
  - However, it may introduce new errors coming from unanticipated shared resource conflicts or contentions

Common SMP Errors

- **Race conditions:**
  - The outcome of the program is dependent on the detailed timing of the threads
- **Deadlock**
  - Threads lock up waiting on a locked resource that will never become available
- **Livelock**
  - Multiple threads working individual tasks which the ensemble can’t finish

Race Conditions

- **The result varies unpredictably based on detailed order of execution for each section**

- **Wrong answers produced without warning**

```
!$OMP PARALLEL SECTIONS
      A=B+C
!$OMP SECTION
      B=A+C
!$OMP SECTION
      C=B+A
!$OMP END PARALLEL SECTIONS
```

OpenMP Death-Traps

- **Are you using thread-safe libraries?**
- **I/O inside a parallel region can interleave unpredictably**
- **Private variables can mask global**
- **Understand when shared memory is coherent. When in doubt use FLUSH**
  - FLUSH provides a means for making memory consistent across threads (only for shared variables).

    !$OMP FLUSH (list)
- **NOWAIT removes implied barriers**
  - Threads can proceed to the next statement without waiting for all other threads to complete the “for” (C) or “do” (Fortran) loop execution

Portable Sequential Equivalence

- **What is Portable Sequential Equivalence (PSE)?**
  - A program is sequentially equivalent if its results are the same with one thread or many threads
  - For a program to be portable (runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored
- **Strong SE: bitwise identical results**
- **Weak SE:equivalent mathematically, not bitwise identical**
  - This is the case for MPI codes

Portable Sequential Equivalence

- **Strong SE:**
  - Locate all cases where a shared variable can be written by multiple threads
    - The access to the variable must be protected
    - If multiple threads combine results into a single value, enforce sequential order

- **Weak SE:**
  - Floating point arithmetic is
    - not associative (results not dependent on grouping of arithmetic, ie (a+b)+c = a+(b+c)) and
    - not commutative (results are not dependent on order of arithmetic, ie a*b = b*a)
  - In most engineering applications, no particular grouping is mathematically preferred so why choose the strong sequential order?

Example

- **The summation of TMP into RES occurs one thread at a time, but in any order so the result is not bitwise equivalent to the sequential one.**

```
!$OMP PARALLEL PRIVATE(I,TMP)
!$OMP DO
          DO I=1,NDIM
            TMP=FOO(I)
!$OMP CRITICAL
            CALL COMBINE(TMP,RES)
!$OMP END CRITICAL
          END DO
!$OMP END PARALLEL
```

Remember that CRITICAL directive allows only one thread (first to arrive) to execute that section.

To be bitwise equivalent, the same thread always must do the sum in the same order.

OpenMP Scalability

- **Memory is often the limit to achievable performance of a shared memory program**
- **On scalable architectures, the latency and bandwidth of memory accesses depend on the locality of those accesses**
- **In achieving good speedup of a shared memory program, data locality is an *essential element***

Data Distribution on DSM Computers

- **Initial data distribution determines on which node of a Distributed Shared Memory machine the memory is placed. This is dependent on**
  - "First touch" or "round-robin" system policies
  - Data distribution directives
  - Explicit page placement
- **Work sharing, e.g., loop scheduling, determines which thread accesses which data**
- **Cache friendliness determines how often main memory is accessed**

False Sharing

- ***Contention*** **is an issue specific to parallel loops, e.g., *false sharing of cache lines***

- ***Cache Friendliness*** **= high locality of references**

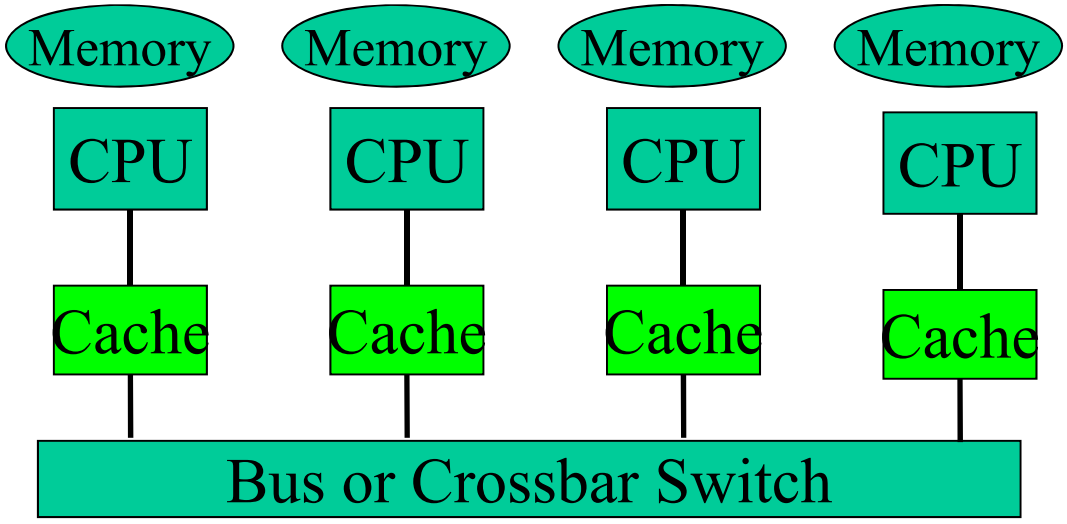
  +

  low contention

SMP- NUMA

- **Memory hierarchies exist in single-CPU computers and Symmetric Multiprocessors (SMPs)**

- **Distributed shared memory (DSM) machines based on Non-Uniform Memory Architecture (NUMA) (like the older SGI Origin) add levels to the hierarchy:**
  - *Local memory* has low latency
  - *Remote memory* has high latency



SGI Origin 2000 Memory

Level	Latency (cycles)
register	0
primary cache	2...3
secondary cache	8...10
local main memory & TLB hit	75
remote main memory & TLB hit	250
main memory & TLB miss	2000
page fault	10^6

TLB: Translation Lookaside Buffer

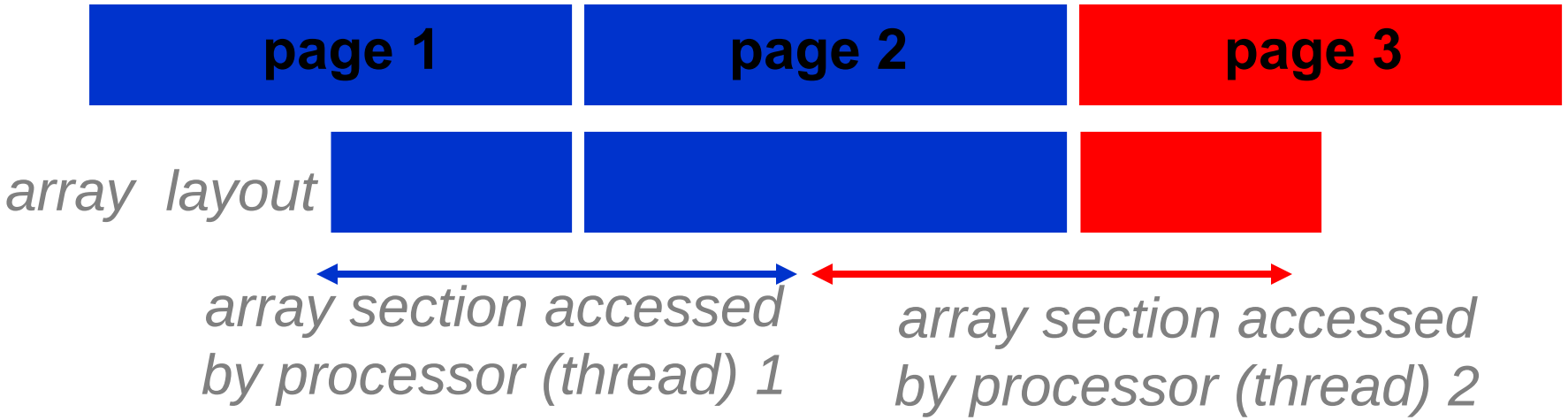
Page Locality

- **An ideal application has full *page locality*:**
  - Pages accessed by a processor are on the same node as the processor, and
  - No page is accessed by more than one processor (no page sharing)

- **Twofold benefit:**
  - Low memory latency
  - Scalability of memory bandwidth

Page Locality Example

- **Consider an array whose size is twice the size of a page, and which is distributed between two processors**
- **Page 1 and page 2 are located on processor 1, page 3 is on processor 2**



- **Page 2 is shared by the two processors, due to the array not starting on a page boundary ➔ page miss**

Page Locality

- **The benefits of page locality are more important for programs that are *not cache friendly***

- **Several data placement strategies for improving page locality include:**
  - System based placement (such as on IRIX)
    - *first-touch*: the process which first references a virtual address causes that address to be mapped to a page on the processor where the process runs
    - *round-robin:* pages allocated to a job are selected from processors traversed in round-robin order
  - Data initialization and directives (system specific, not in the OpenMP standard)

Using an SPMD Approach Instead of Loop Breakup

- **An SPMD (single program multiple data) approach can be used in both Distributed- and Shared-Memory Systems**
  - In the context of openmp, this pertains to placing a subroutine in a parallel section rather than breaking loops up
- **Data is distributed explicitly among processes**
- **With message passing, e.g., MPI, where no data is shared, data is explicitly communicated**
  - Synchronization is explicit or embedded in communication
- **With parallel regions in OpenMP, both SPMD and data sharing are supported**

Using an SPMD Approach Instead of Loop Breakup

- **One can achieve a potentially higher parallel fraction with an SPMD approach rather than with loop parallelism in Shared-Memory Systems**
  - The fewer parallel regions, the less overhead
  - However, more explicit synchronization needed than for loop parallelization
    - Essentially using OpenMP to mimic the same type of parallelization strategy that would be used with MPI
  - Does not promote incremental parallelization and requires manually assigning data subsets to threads

SPMD Example: Matrix Initialization

- **A single parallel region, no scheduling needed, each thread explicitly determines its work**
  - **threads are independent**

```
  program mat_init
  implicit none
  integer, parameter::N=1024
  real A(N,N)!A and N are shared
  integer :: iam, np
  iam = 0
  np = 1
!$omp parallel private(iam,np)
  np = omp_get_num_threads()
  iam = omp_get_thread_num()
! Each thread calls work
  call work(N, A, iam, np)
!$omp end parallel
  end
```

```
subroutine work(n, A, iam, np)
integer n, iam, n
real A(n,n)
integer :: chunk,low,high,i,j
chunk = (n + np - 1)/np
low = 1 + iam*chunk
high=min(n,(iam+1)*chunk)
! Note that both loops are
!   parallelized with SPMD approach
do j = low, high
  do I=1,n
   A(I,j)=sqrt(real(i*i+j*j))
  enddo
enddo
return
```

Summary of OpenMP Constructs

- **Parallel region**

```
!$omp parallel          #pragma omp parallel
```

- **Worksharing**

```
!$omp do                #pragma omp for
!$omp sections          #pragma omp sections
!$single                #pragma omp single
!$workshare             #pragma workshare
```

- **Data environment**
  - directive: threadprivate
  - clauses: shared, private, lastprivate, reduction, copyin, copyprivate
- **Synchronization**
  - directives: critical, barrier, atomic, flush, ordered, master
- **Runtime functions/environment variables**

Achieving Good Parallel Performance

- **Optimize single-CPU performance**
  - Maximize cache reuse
  - Eliminate cache misses
  - Use compiler flags to optimize when possible
- **Parallelize as high a fraction of the work as possible with OpenMP**
  - Preserve cache friendliness
  - Avoid synchronization and scheduling overhead:
    - partition in few, large parallel regions,
    - avoid reduction, single and critical sections,
    - use static scheduling
    - partition work to achieve load balancing
- **Check correctness of parallel code**
  - Run OpenMP compiled code first on one thread, then on several threads

OpenMP or MPI?

- **Do you need total portability?**
  MPI

- **Do you need to use hundreds of processors?**
  MPI

- **Do you have access to DSM memory machines?**
  OpenMP by itself or in conjunction with MPI

- **Do you want to be ready for the next generation of computers?**
  MPI and OpenMP or
  MPI and OpenACC (for GPUs) and OpenMP (for multi-cores/node)

OpenMP on hpc1

- **There are a couple of ways to perform shared-memory parallel computing of loops on hpc1 using the pgf90 compiler**
  - You can read about the various options by reading the manpages of pgf90:

    man pgf90 (or by reading the users' manual)
  - Compiler options include:
    - The –concur=option[altcode, noaltcode, dist:block, dist:cyclic, cncall, noassoc] (See the manpage for description of options)
    - The –Mconcur option must be used in linking step.
  - OpenMP can be invoked:
    - Add the –mp option during compilation and linking (see chapter 10 of PGI users' manual)
    - The –noopenmp option will turn off any OpenMP directives so that you can run serial loops
  - You can try these out on any of your project simulation codes (project-1, -3, or -5)

Manuals and Other Information

- **I have put the following manuals out on smartsite under "Additional Material/":**
- **Under Compiling**
  - pgf90, gnu, intel manuals
- **Under OpenMP**
  - OpenMP manuals

OpenMP on hpc1

- **Compiling:**

  /share/apps/pgi-2015/linux86-64/15.4/bin/pgf90 –c –mp file.f

- **Linking:**

  /share/apps/pgi-2015/linux86-64/15.4/bin/pgf90 –o exec –mp file.o -lpthread

Homework 8

- **Read Chaps 9 and 11 in Introduction to Parallel Computing by Grama et al.**

Shared Memory Parallel Computing using Graphical Processing Units (GPUs) - Hardware Considerations

- Graphical Processing Units (GPUs) have been used extensively to perform parallel kernels in video games

- Recent developments have shown that they can also be used for scientific simulations

- GPUs are different than OpenMP shared-memory parallelization, however, and you need to understand those differences in order to take advantage of the GPUs

The Dilemma

- **Parallelization is essential in modern engineering physics simulations**
- **Traditional Thinking Says:**
  - Just add more CPUs!

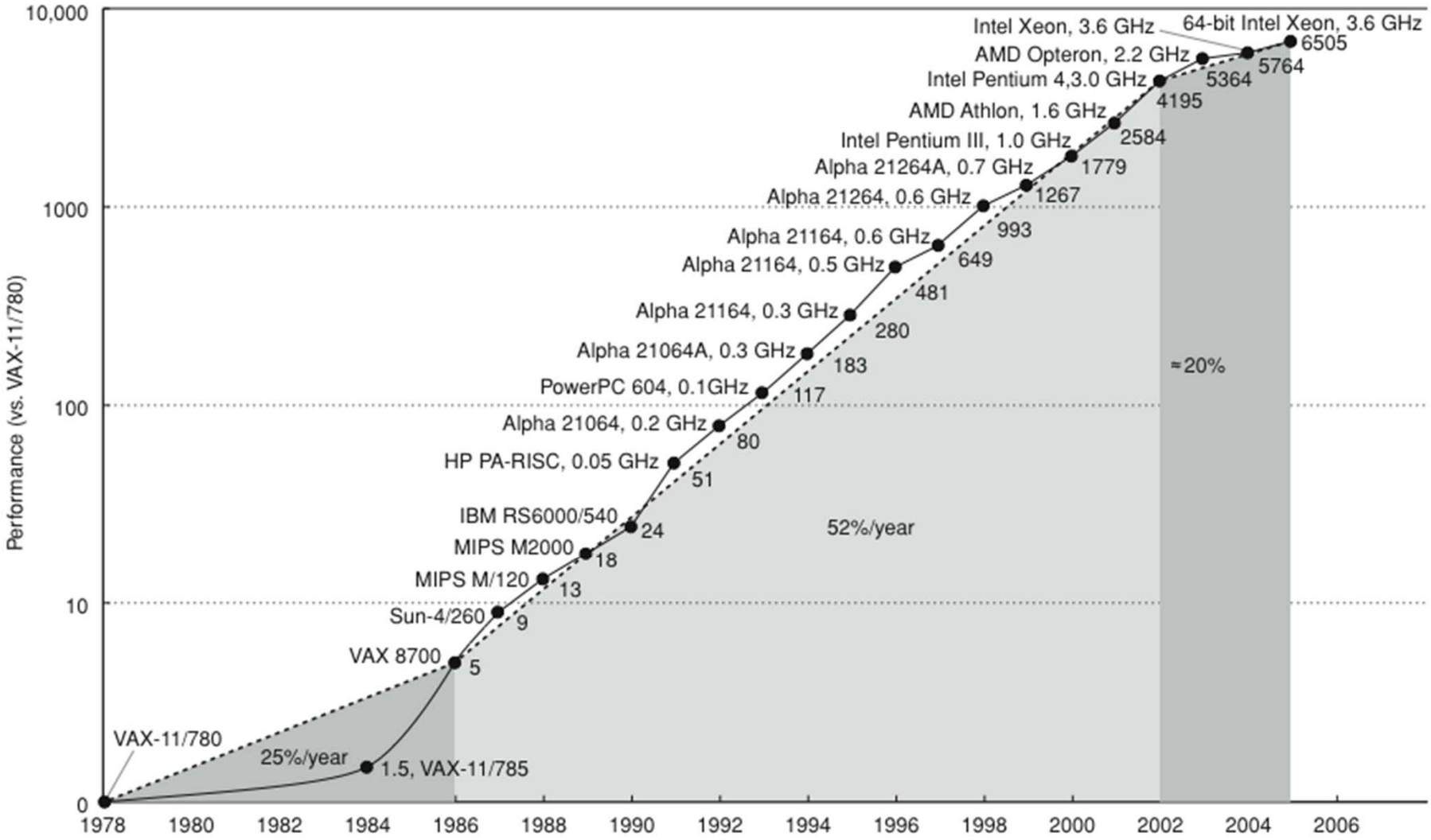
    (or more recently….
  - Just add more Cores!

    and increase parallelization
- **Problem with this Thinking is:**
  - Cost ! Even though prices are dropping, we will unlikely reach an affordable system with CPU/Cores in near future
  - Recent CPU price

    quad-core, 16G RAM is ~$1700 (build yourself), $2500 (built for you)

    dual quad-core, 32G RAM is $2400 (build yourself), $4000 (built for you)
  - 500 core cluster → $150,000 (build yourself) to $312,000 (built for you) + $racks + $switches + $power + $space

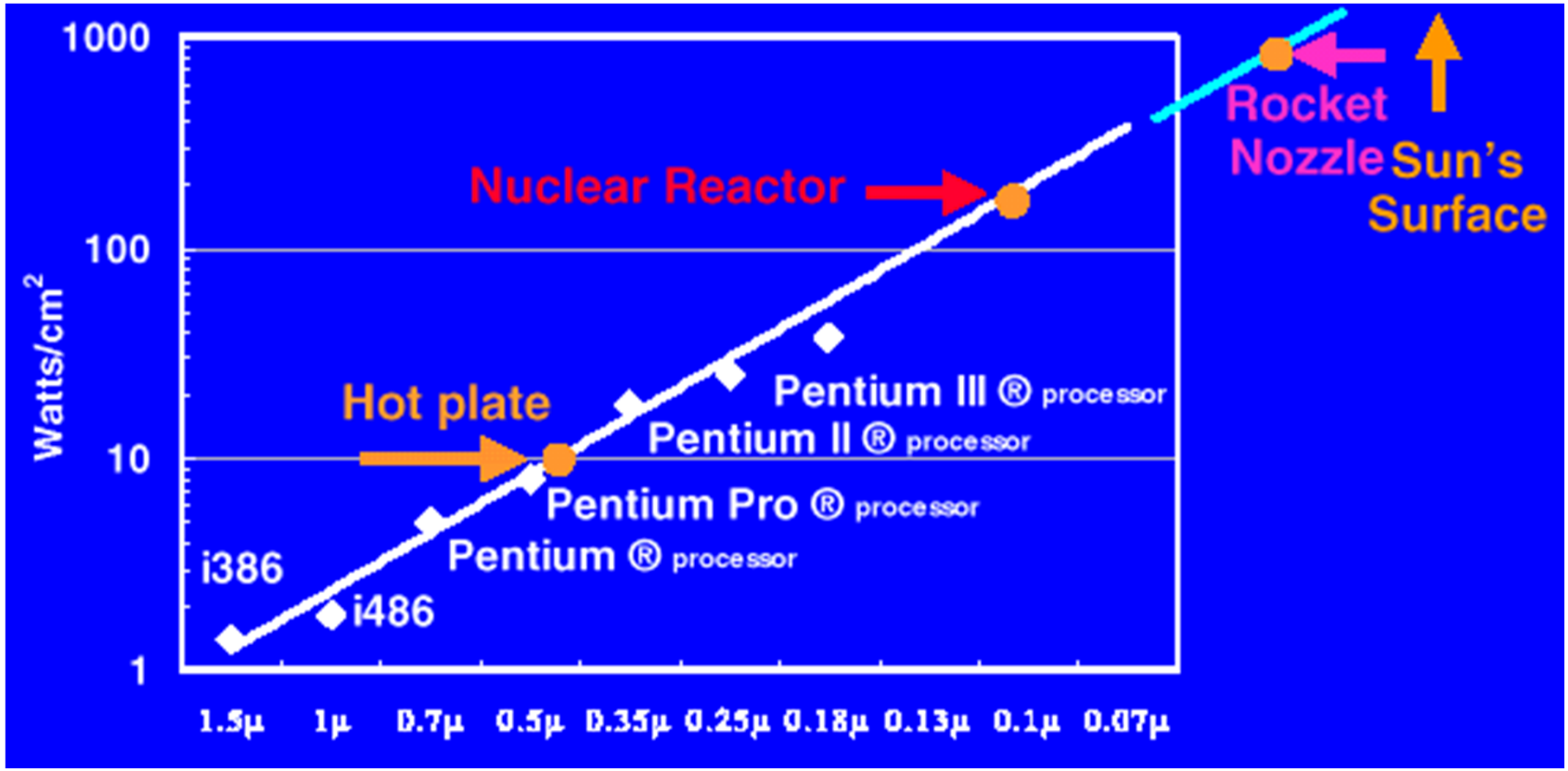
“Toping Out” of the CPU



[H&P Figure 1.1]

Power Limits Performance

- **Cost in the form of power requirements is also a problem for large CPU clusters**



[courtesy of Bob Colwell]

The Computer Footprint Problem

- **Space requirements for multi-hundred node CPU/Core systems can be hard to obtain**

- **Air-conditioning requirements are also large adding onto power requirements**

Motivation

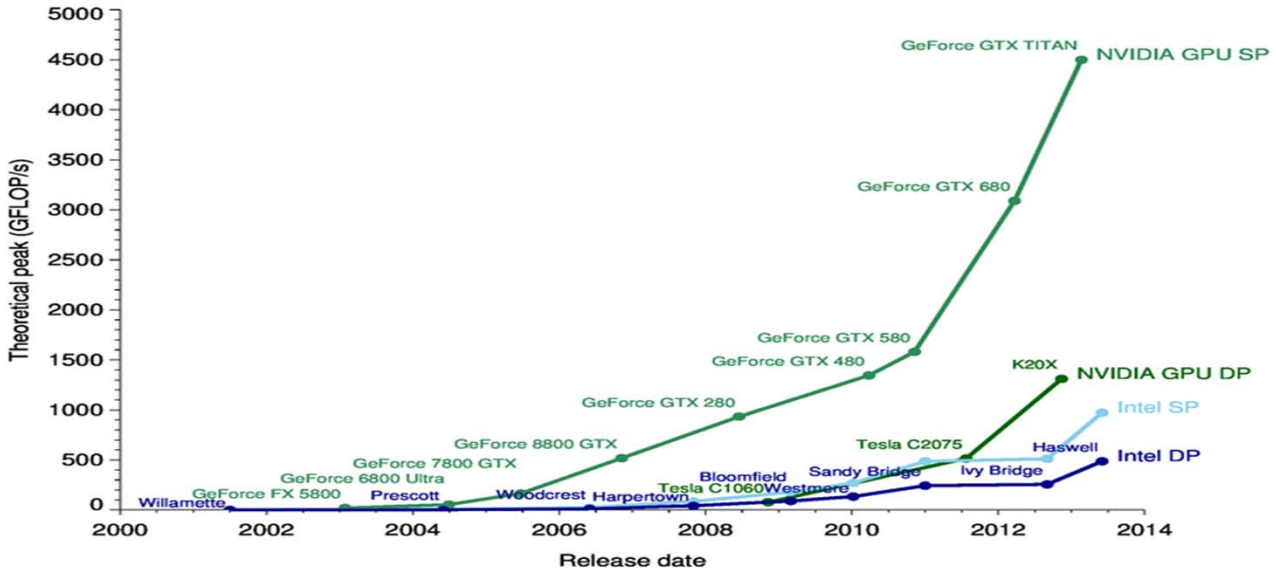
- **Graphical processing units (GPUs) have proven success for gaming applications**
- **Recently shown to also be useful for scientific simulations**
- **GPU Costs:**
  - ~$500 for 128 floating-point units
  - Example: Our GPU cluster in ECE
    - 8 nodes of single quad-cores (32 cores)
    - 1 GPU per core → 32 GPUs
    - 12 Teraflops of peak performance, ~$25,000-$30,000
  - Low space and power requirements

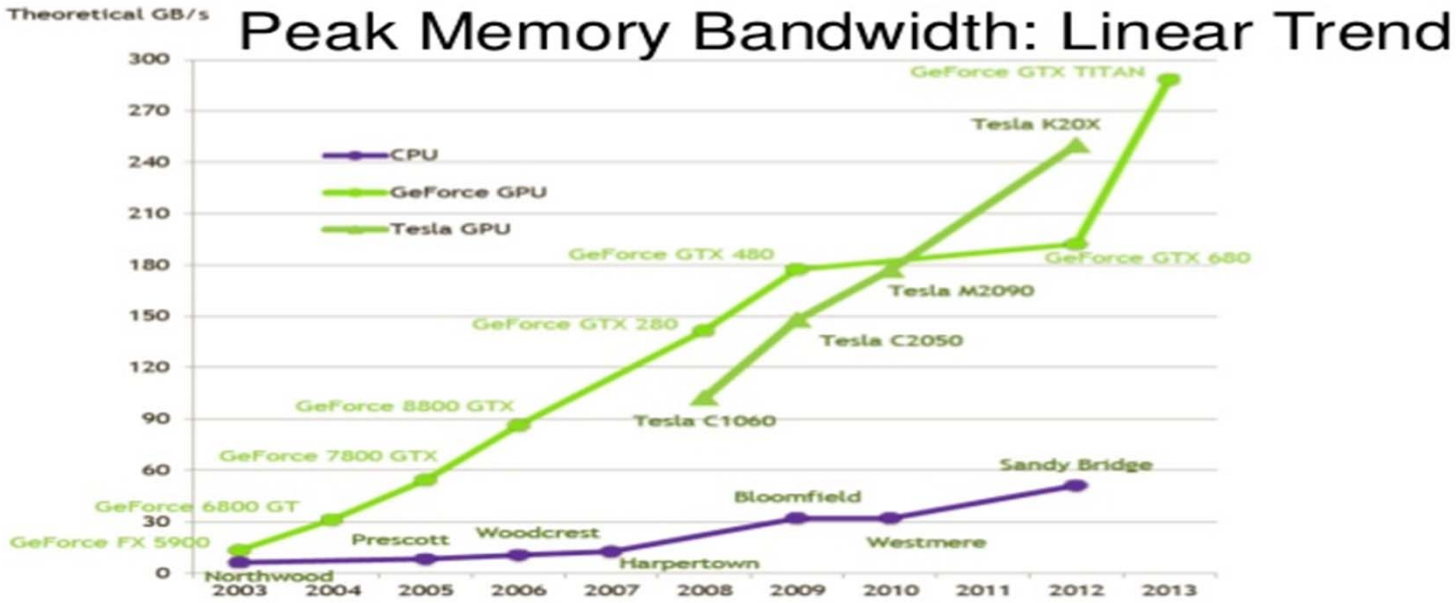
- **Cost Effective Means of Achieving our Goals!**

Advantages of Graphical Processors

- **Order of magnitude increase in**
  - floating point
  - memory bandwidth
- **Very low cost**
- **Easy to program with new programming models (CUDA-C or CUDA-Fortran)**
- **Good at processing large data sets where same operation is applied over large arrays**
- **Scales well when added to cluster nodes**
- **Perfect fit for some CFD and other engineering applications (especially explicit algorithms)**

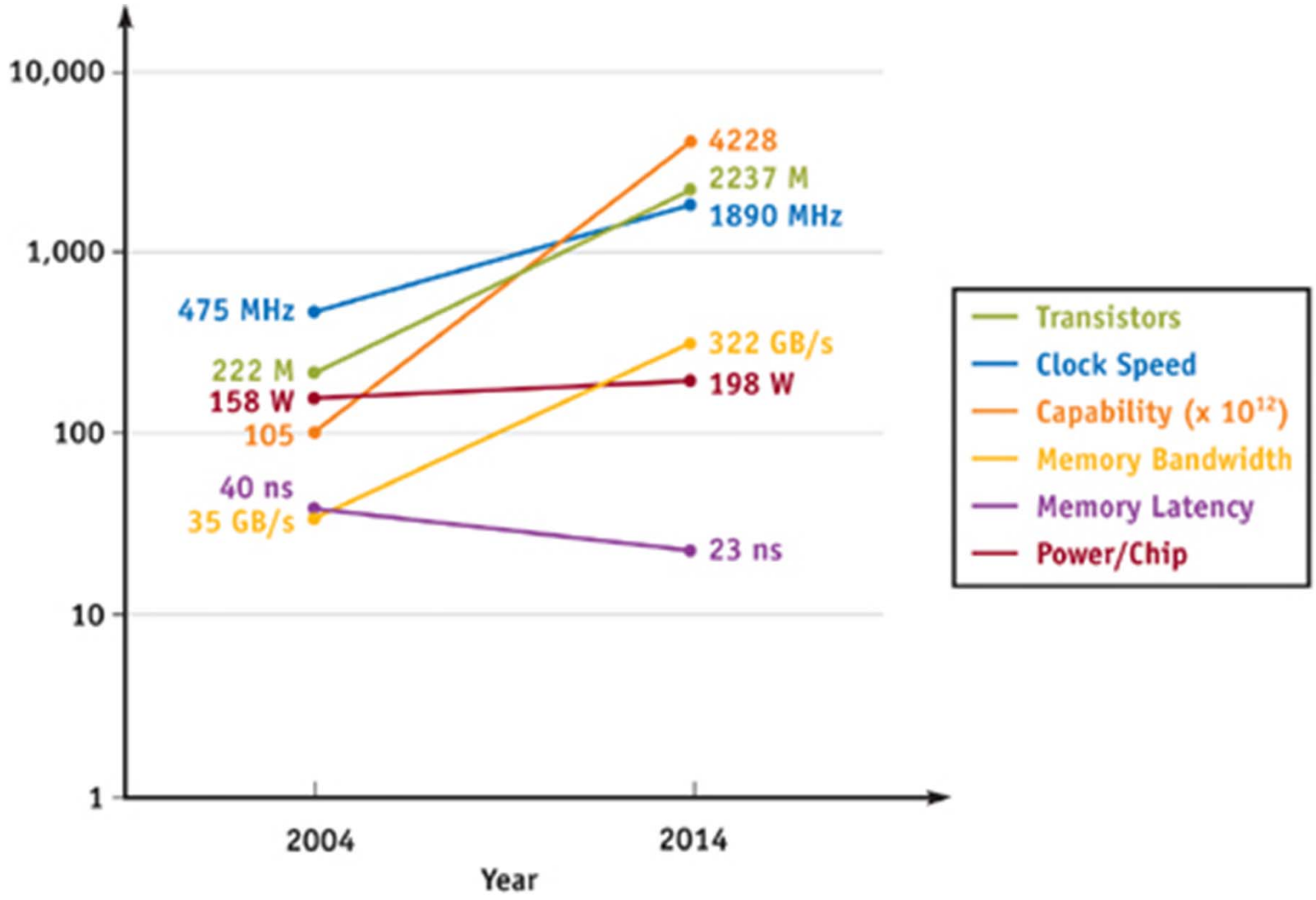
GPU vs CPU Performance Trends





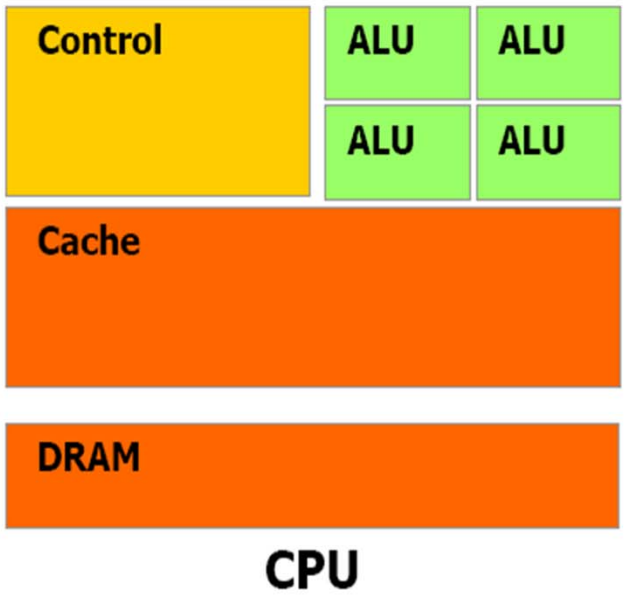
Figures courtesy NVIDIA
(http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf)

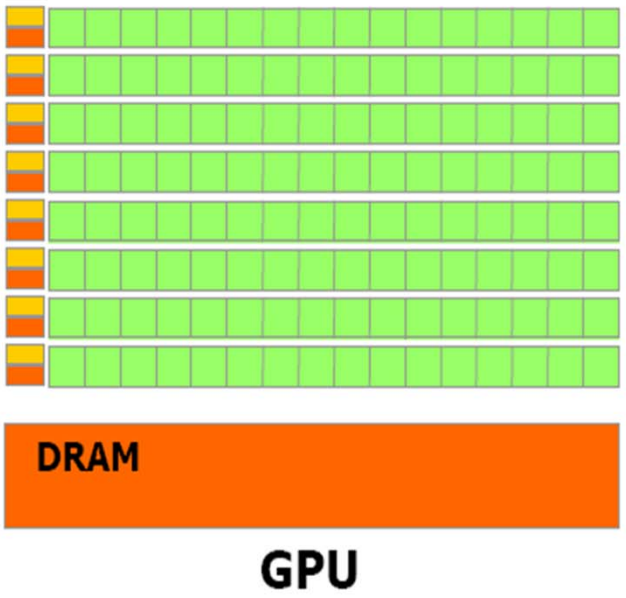
Forecast of GPU Performance



GPU Architecture

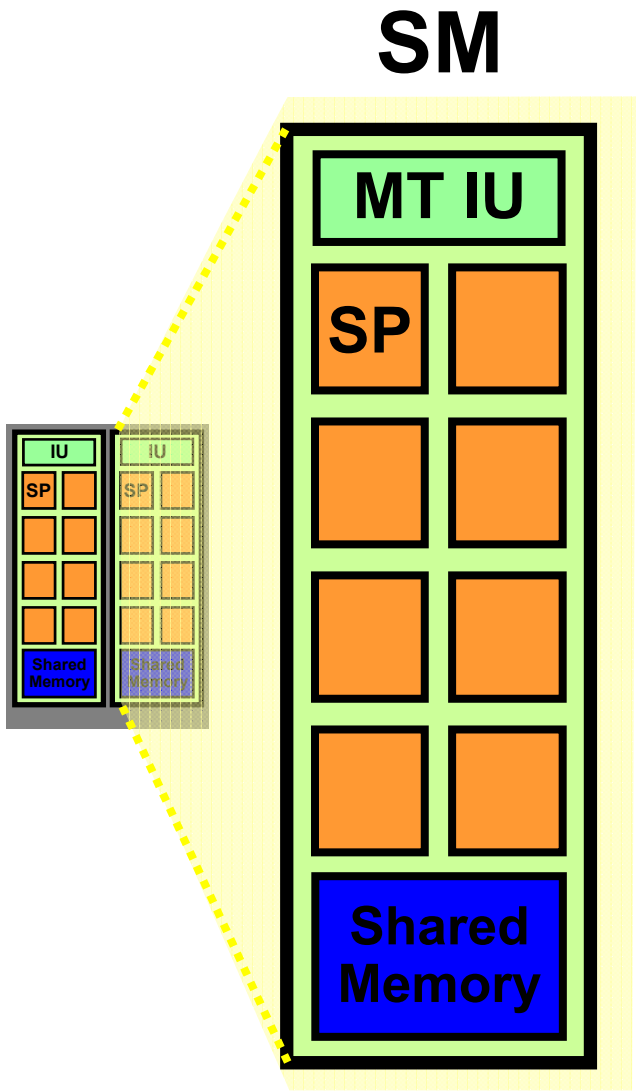
- **More transistors devoted to data processing** (shown in green)
- **Optimized for throughput**
- **Data Parallel – Same program executed on many data elements in parallel (SPMD)**





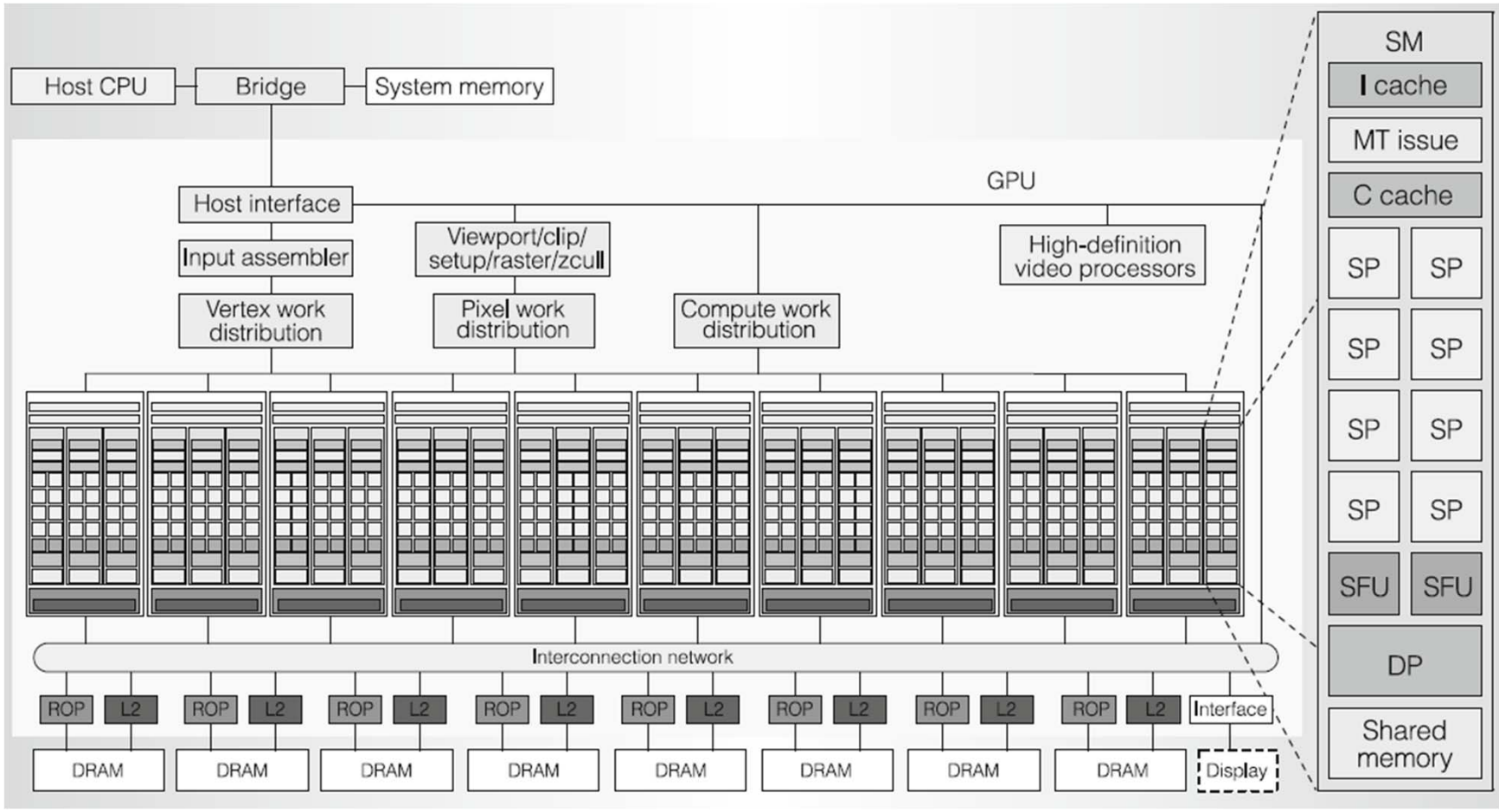
Figures courtesy NVIDIA
(http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf)

GeForce SM Multithreaded Multiprocessor



- **Each SM processor runs a block of threads**
- **SM has 8 SP Thread Processors**
  - 32 GFLOPS peak at 1.35 GHz
  - IEEE 754 32-bit floating point
- **Scalar ISA bus**
- **Up to 768 threads, hardware multi-threaded**
- **16KB Shared Memory**
  - Concurrent threads share data
  - Low latency load/store

GeForce GPU Architecture: GT200



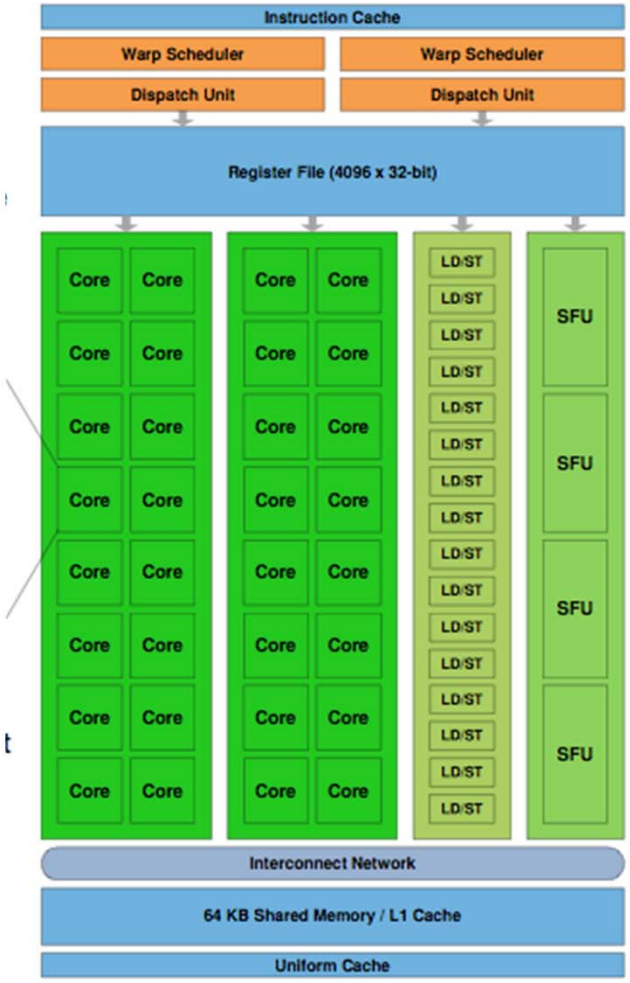
Figures courtesy NVIDIA

Latest Tesla C-Series Workstation GPUs



	Tesla C1060	Tesla C2050	Tesla C2070
Architecture	Tesla 10-series GPU	Tesla 20-series GPU	
Number of Cores	240	448	
Caches	16 KB Shared Memory / 8 cores	64 KB L1 cache + Shared Memory / 32 cores, 768 KB L2 cache	
Floating Point Peak Performance	933 Gigaflops (single) 78 Gigaflops (double)	1030 Gigaflops (single) 515 Gigaflops (double)	
GPU Memory	4 GB	3 GB 2.625 GB with ECC on	6 GB 5.25 GB with ECC on
Memory Bandwith	102 GB/s (GDDR3)	144 GB/s (GDDR5)	
System I/O	PCIe x16 Gen2	PCIe x16 Gen2	
Power	188 W (max)	247 W (max)	225 W (max)
Available	Available now	Shipping in May	Q3 2010

Tesla C2050/C2070 – Fermi Architecture

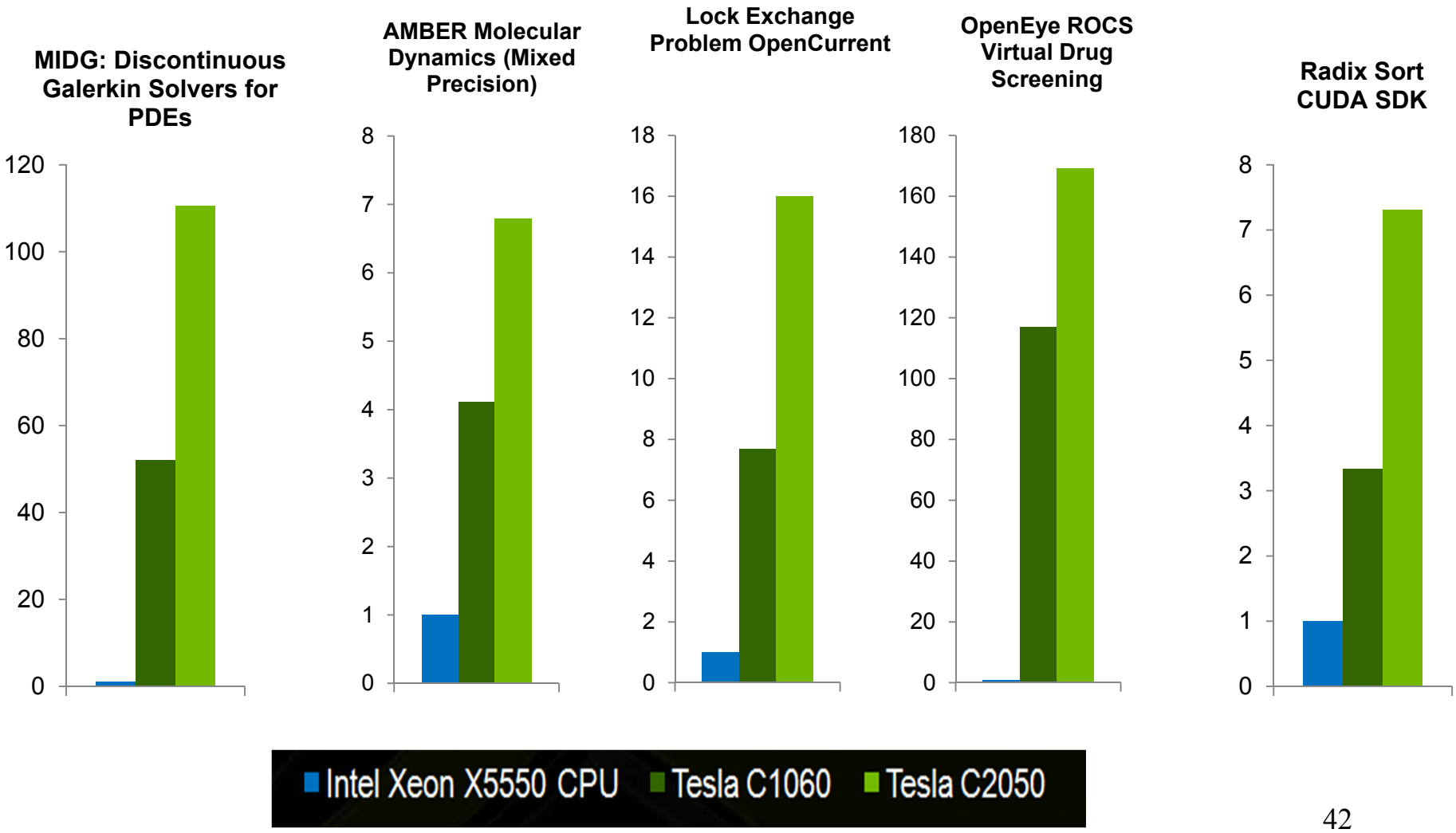


Fermi Streaming Multiprocessor (SM)

- 448 ALUs “cores”
- Up to 448 threads running simultaneously.
- 14 SM processors, 32 ALUs each
- 1 Warp is 32 threads
- 3 GB RAM (“Global Memory”)
- Global Memory operations are scheduled per Warp (Fermi)
- Threads can not communicate
- Threads in an individual thread-block can access all shared memory information.
- Registers are fastest, but maximum 63 registers/thread.
- Extra kernel registers stored in global memory (called spillage).
- Our cluster “Matrx” currently has 12 GPUs spread across 3 nodes (4 each).

Source: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

Latest Performance Summary



Preliminary data

GPU Programming: A Highly Multi-threaded Co-Processor

- **The GPU is viewed as a compute device that:**
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in shared-memory parallel

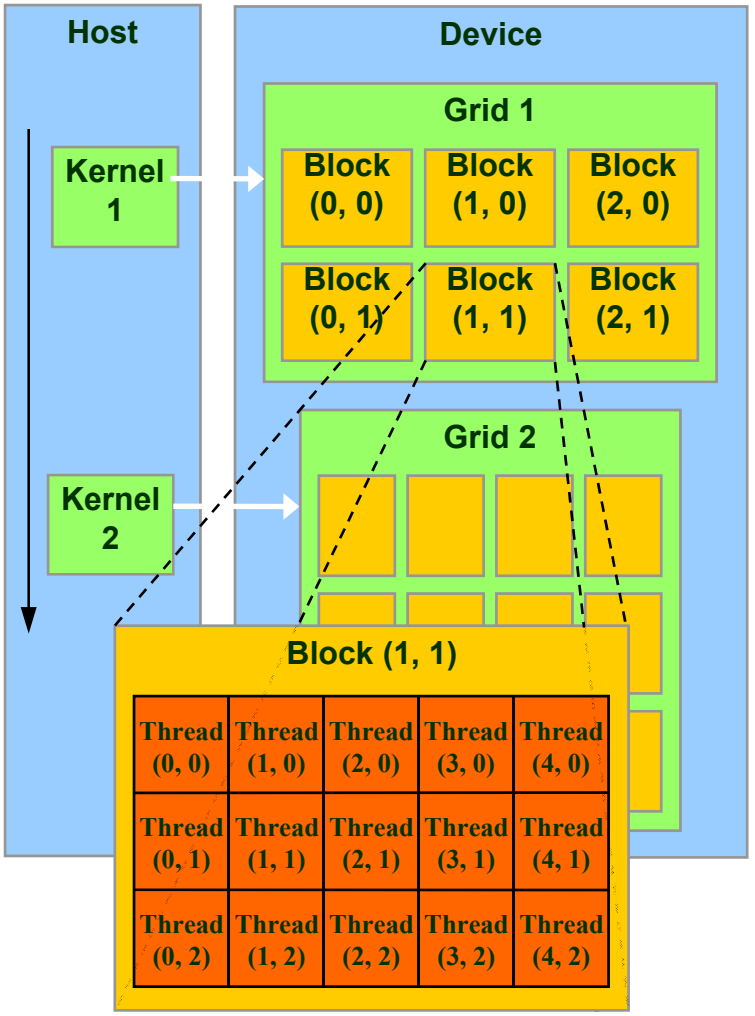
- **Data-parallel portions of an application execute on the device as kernels that run many cooperative threads in parallel**

Programming Model: A Highly Multi-threaded Co-Processor

- **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

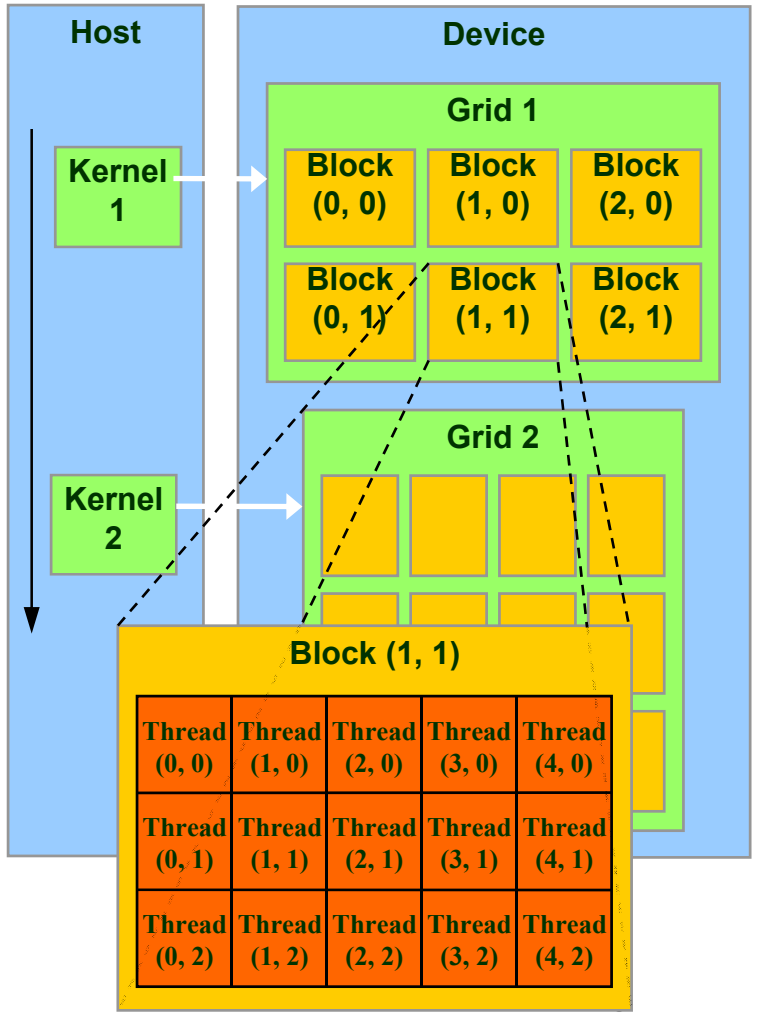
Programming Model (SPMD + SIMD): Thread Batching

- **An algorithm kernel is executed as a grid of thread blocks**
  - Kernels are launched in grids
    - One kernel executes at a time
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Efficiently sharing data through shared memory
  - Synchronizing their execution
    - For hazard-free shared memory accesses



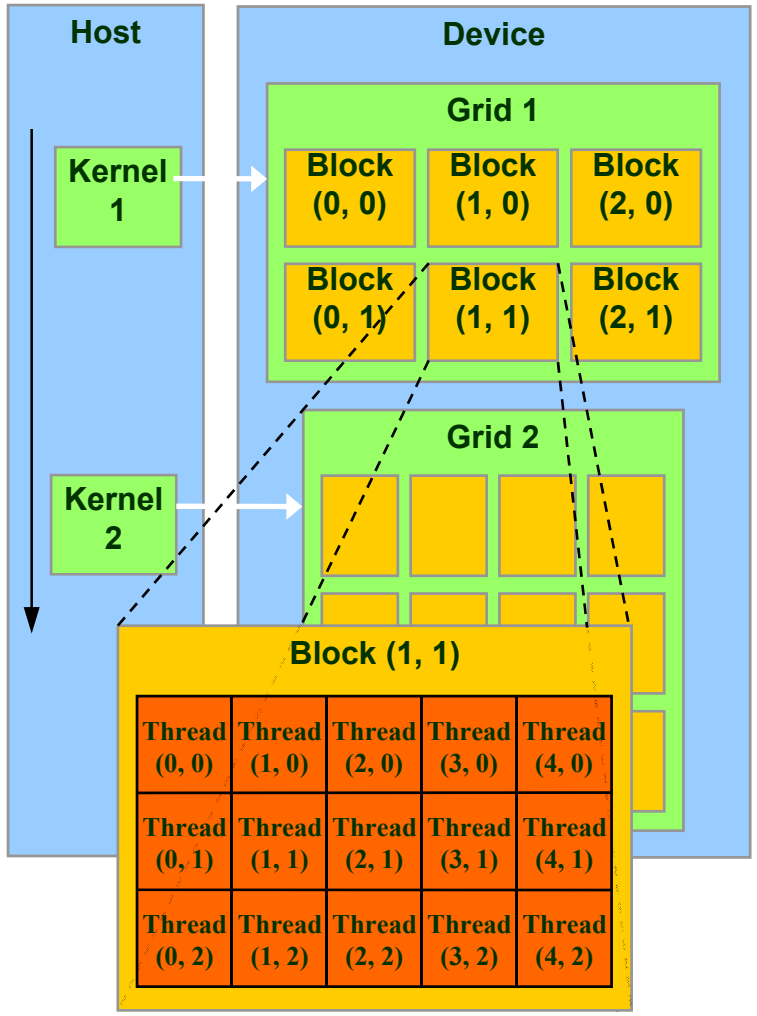
Execution Model

- **A block executes on one multiprocessor**
  - Does not migrate
  - Two threads from two different blocks cannot cooperate
    - Blocks are independent



Execution Model

- **Several blocks can reside concurrently on one multiprocessor (SM)**
  - Control limitations (of G8X/G9X GPUs):
    - At most 8 concurrent blocks per SM
    - At most 768 concurrent threads per SM
  - Number is further limited by SM resources
    - Register file is partitioned among all resident threads
    - Shared memory is partitioned among all resident thread blocks



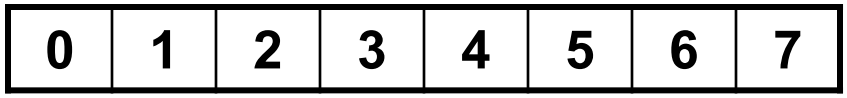
The CUDA Abstraction

- **Threads have arbitrary access to memory**
  - Threads within a thread block share 16 KB shared memory on chip
- **CPU and GPU have separate memory spaces**
  - Data is moved across PCIe bus
  - Use function to allocate/set/copy memory on GPU
- **Host (CPU) manages device (GPU) memory**
- **CUDA maps thread blocks to hardware**
- **Programmer responsible for CPU←→GPU communication and synchronization**

CUDA Kernels and Threads

- **Parallel portions of an application are executed on the device as kernels**
  - One kernel is typically executed at a time
  - A “Fermi” GPU can run two kernels simultaneously using CUDA streams.
  - Memory copies to host can occur simultaneously with kernel launch (Fermi)
- **A CUDA kernel is executed by an array of threads**
  - All threads run the same code except when using CUDA streams
  - Each thread has an ID that it uses to compute memory addresses and make control decisions

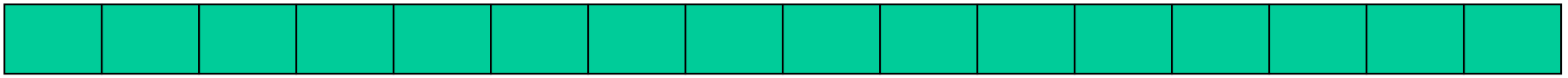
threadID



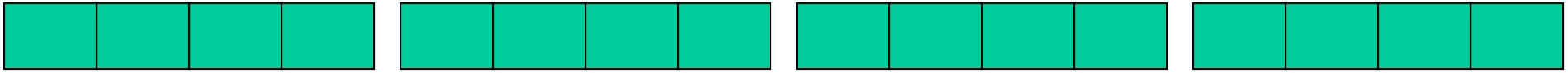
From NVIDIA: Volume I

Example: Increment Array Element

- **Increment N-element vector a by scalar b**



- **Let's assume N=16, blockDim=4 → 4 blocks**



blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3
blockDim.x=4	blockDim.x=4	blockDim.x=4	blockDim.x=4
threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3	threadIdx.x=0,1,2,3
idx=0,1,2,3	idx=4,5,6,7	idx=8,9,10,11	idx=12,13,14,15

Int idx = blockDim.x*blockId.x+threadIdx.x

Will map from local index threadIdx to global index

NB: blockDim should be >=32 in real code, this is just an example

From NVIDIA: Volume I

2D or 3D Kernel Launches

- CUDA can also launch kernels according to a 2D or 3D grid structure
- This corresponds to multi-dimensional arrays
- Programmer can decide shape of thread-block
- Example: Block can be 32 (i-direction) * 4 (j-direction) for 128 threads per block in 2D grid structure.
- Memory fetches occur per warp or half warp (depending on GPU), so memory coalescing is important.
- Pointers (stored on GPU) can be used to construct multi-dimensional array structure
- Each thread figures out its own i,j location:

  `int i = blockDim.x * blockIdx.x + threadIdx.x;`

  `int j = blockDim.y * blockIdx.y + threadIdx.y;`
- Access array on GPU as T[j][i].

The Challenge

- **Taking advantage of the GPU:**
  - How can you “retrofit” existing codes to use GPUs as an option?
  - What level of speed-ups are obtainable? (embarrassingly parallel problems maximize speed-ups)
  - Is it worth the effort?
  - Are GPUs always advantageous?

- **These are the issues we are most interested in as engineers**
  - Do we have to completely re-write our software?
  - What are the penalties for not doing so?