Mestrado em Informática

2010/11

A.J.Proença

Tema

Arquitecturas Paralelas (2)

Adaptado de
Documentos da NVidia e de slides de outras apresentações

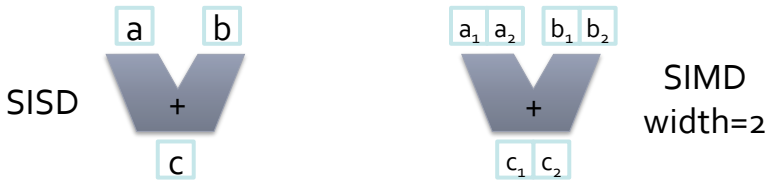
Estrutura do tema AP

1. A evolução das arquitecturas pelo paralelismo
2. Multiprocessadores (SMP e MPP)
3. Data Parallelism: SIMD, Vector, GPU, …
4. Topologias de interligação

- An alternate classification

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors



- Single Instruction Multiple Data architectures make use of data parallelism
- SIMD can be area and power efficient
  - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler

- Operate element-wise on vectors of data
  - E.g., MMX and SSE instructions in x86
    - Multiple data elements in 128-bit wide registers
- All processors execute the same instruction at the same time
  - Each with different data address, etc.
- Simplifies synchronization
- Reduced instruction control hardware
- Works best for highly data-parallel applications

- Highly pipelined function units
- Stream data from/to vector registers to units
  - Data collected from memory into registers
  - Results stored from registers to memory
- Example: Vector extension to MIPS
  - 32 × 64-element registers (64-bit elements)
  - Vector instructions
    - `lv`, `sv`: load/store vector
    - `addv.d`: add vectors of double
    - `addvs.d`: add scalar to each element of vector of double
- Significantly reduces instruction-fetch bandwidth

Vector Code Example

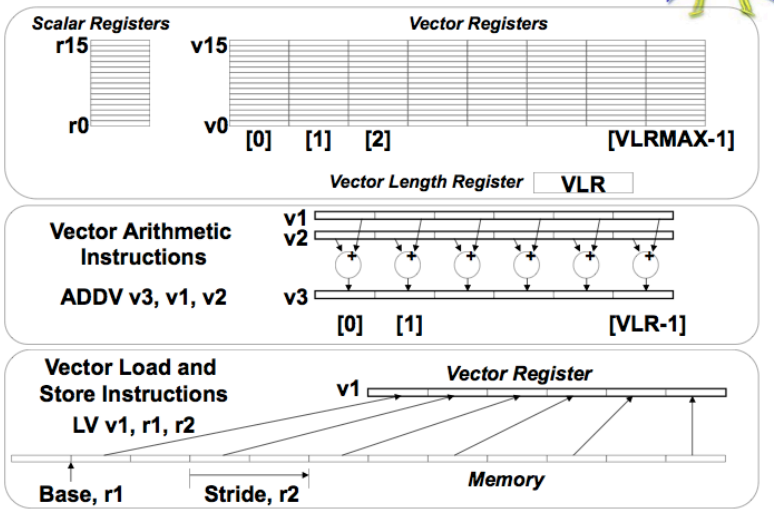
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# Scalar Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

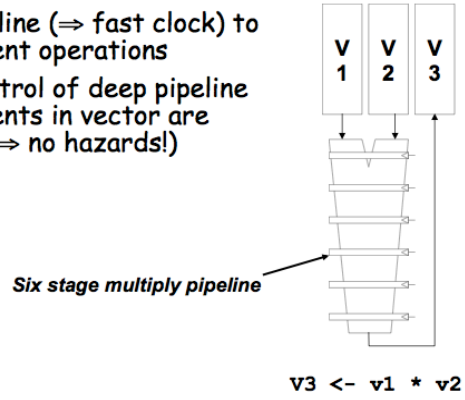
- Require programmer (or compiler) to identify parallelism
  - Hardware does not need to re-extract parallelism
- Many multimedia/HPC applications are natural consumers of vector processing

Vector Programming Model



Vector Arithmetic Execution

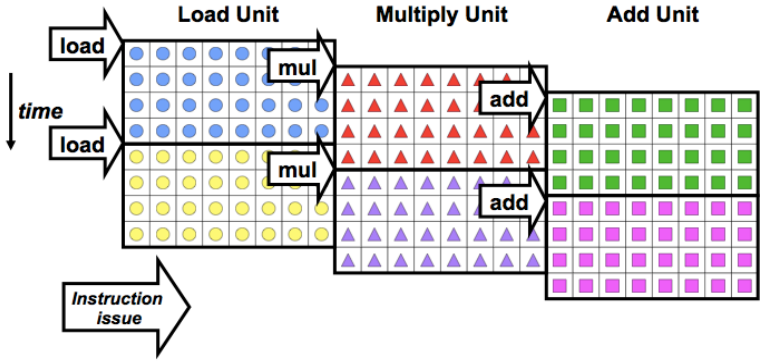
- Use deep pipeline (⇒ fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (⇒ no hazards!)



Vector Instruction Parallelism

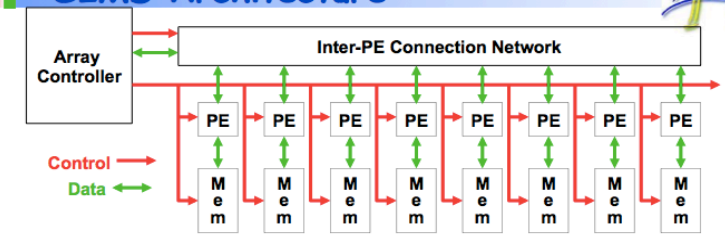
Can overlap execution of multiple vector instructions

- Consider machine with 32 elements per vector register and 8 lanes:



Complete 24 operations/cycle while issuing 1 short instruction/cycle

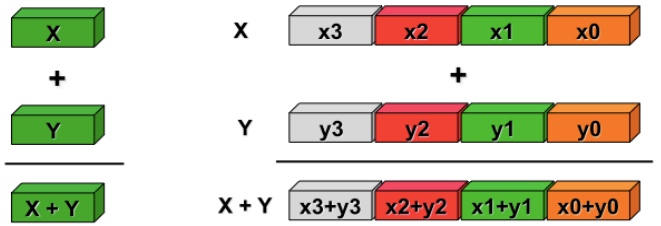
SIMD Architecture



- Single Instruction Multiple Data (SIMD)
- Central controller broadcasts instructions to multiple processing elements (PEs)
  - Only requires one controller for whole array
  - Only requires storage for one copy of program
  - All computations fully synchronized
- Recent Return to Popularity:
  - GPU (Graphics Processing Units) have SIMD properties
  - However, also multicore behavior, so mix of SIMD and MIMD (more later)
- Dual between Vector and SIMD execution

Pseudo SIMD: (Poor-Man's SIMD?)

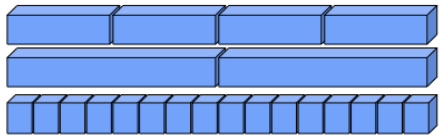
- Scalar processing
  - traditional mode
  - one operation produces one result
- SIMD processing (Intel)
  - with SSE / SSE2
  - one operation produces multiple results



Slide Source: Alex Klimovitski & Dean Macri, Intel Corporation

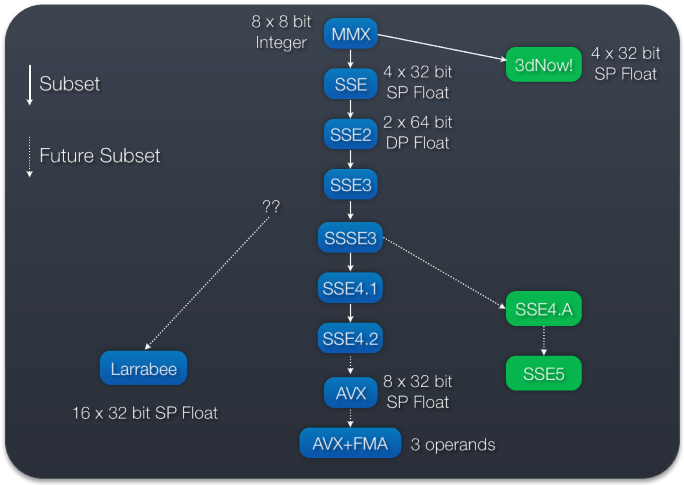
E.g.: SSE / SSE2 SIMD on Intel

- SSE2 data types: anything that fits into 16 bytes, e.g.,

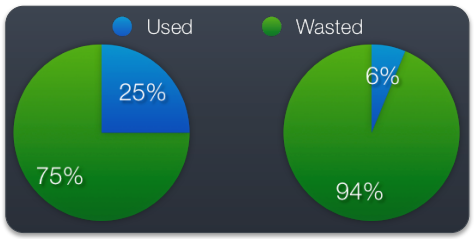


- Instructions perform add, multiply etc. on all the data in this 16-byte register in parallel
- Challenges:
  - Need to be contiguous in memory and aligned
  - Some instructions to move data from one part of register to another
- In theory, the compiler understands all of this
  - When compiling, it will rearrange instructions to get a good "schedule" that maximizes pipelining, uses FMAs and SIMD
  - It works with the mix of instructions inside an inner loop or other block of code
- But in practice the compiler may need your help

A Brief History of x86 SIMD



What to do with SIMD?



4 way SIMD (SSE) 16 way SIMD (LRB)

- Neglecting SIMD in the future will be more expensive
  - AVX: 8 way SIMD, Larrabee: 16 way SIMD, Nvidia: 32 way SIMD, ATI: 64 way SIMD
- This problem composes with thread level parallelism
- We need a programming model which addresses both problems

Vector vs. Scalar

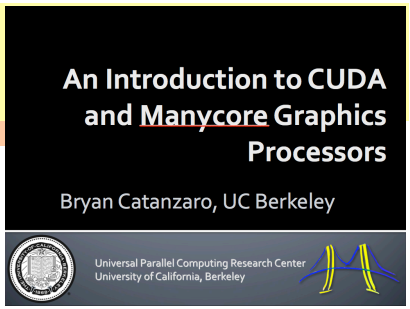
- Vector architectures and compilers
  - Simplify data-parallel programming
  - Explicit statement of absence of loop-carried dependences
    - Reduced checking in hardware
  - Regular access patterns benefit from interleaved and burst memory
  - Avoid control hazards by avoiding loops
- More general than ad-hoc media extensions (such as MMX, SSE)
  - Better match with compiler technology

- Processing is highly data-parallel
  - GPUs are highly multithreaded
  - Use thread switching to hide memory latency
    - Less reliance on multi-level caches
  - Graphics memory is wide and high-bandwidth
- Trend toward general purpose GPUs
  - Heterogeneous CPU/GPU systems
  - CPU for sequential code, GPU for parallel code
- Programming languages/APIs
  - DirectX, OpenGL
  - C for Graphics (Cg), High Level Shader Language (HLSL)
  - Compute Unified Device Architecture (CUDA)

- Don't fit nicely into SIMD/MIMD model
  - Conditional execution in a thread allows an illusion of MIMD
    - But with performance degredation
    - Need to write general purpose code with care

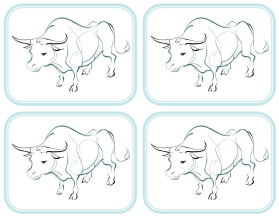
	Static: Discovered at Compile Time	Dynamic: Discovered at Runtime
Instruction-Level Parallelism	VLIW	Superscalar
Data-Level Parallelism	SIMD or Vector	**Tesla Multiprocessor**

Overview



- Terminology: Multicore, Manycore, SIMD
- The CUDA Programming model
- Mapping CUDA to NVidia GPUs
- Experiences with CUDA

Multicore and Manycore

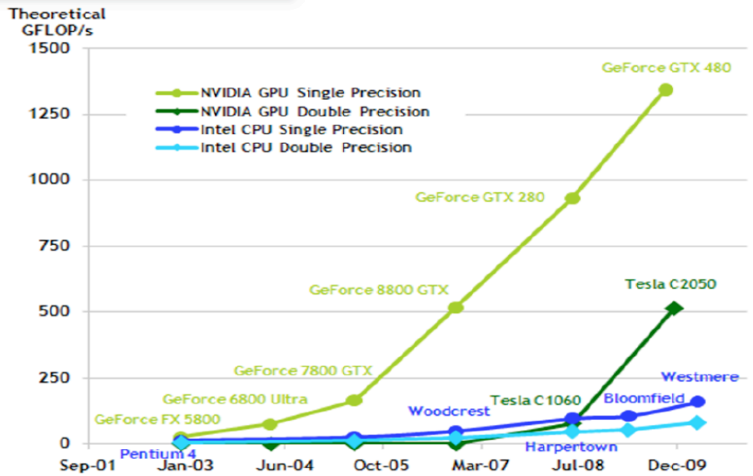


Multicore

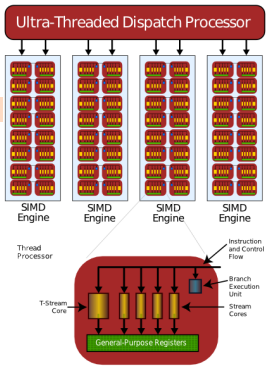
Manycore

- Multicore: yoke of oxen
  - Each core optimized for executing a single thread
- Manycore: flock of chickens
  - Cores optimized for aggregate throughput, deemphasizing individual performance

Performance gap between GPUs and CPUs



What is a core?



- Is a core an ALU?
  - ATI claims 800 streaming processors!!
    - 5 way VLIW * 16 way SIMD * 10 “SIMD cores”
- Is a core a SIMD vector unit?
  - NVidia claims 448 streaming processors!!
    - 32 way SIMD * 14 “multiprocessors”
      - To match ATI, they could count another factor of 2 for dual issue
- In these slides, we use core consistent with the CPU world
  - Superscalar, VLIW, SIMD are part of a core’s architecture, not the #cores

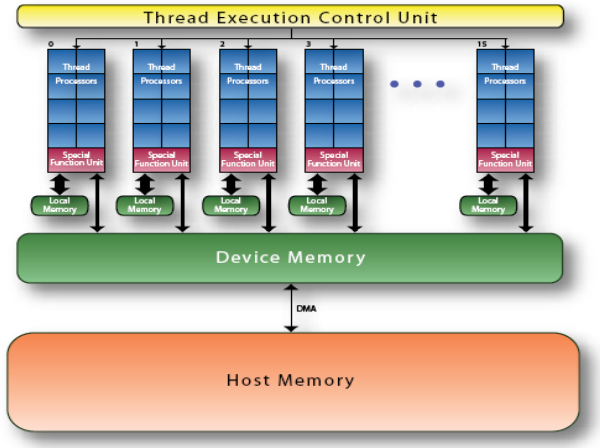
The CUDA Programming Model

- ***C**ompute **U**nified **D**evice **A**rchitecture*
- CUDA is a recent programming model, designed for
  - Manycore architectures
  - Wide SIMD parallelism
  - Scalability
- CUDA provides:
  - A thread abstraction to deal with SIMD
  - Synchr. & data sharing between small groups of threads
- CUDA programs are written in C with extensions
- OpenCL inspired by CUDA, but hw & sw vendor neutral
  - Programming model essentially identical

CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device
- Data-parallel portions of an application are expressed as device kernels which run on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

A view of a GPU as a compute device: the G80

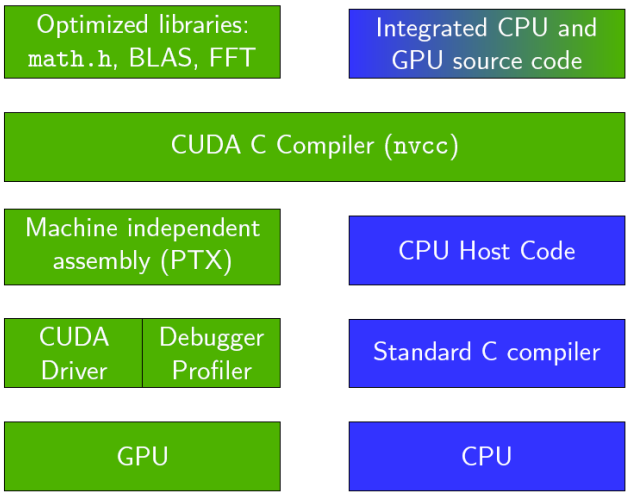


Programming Model

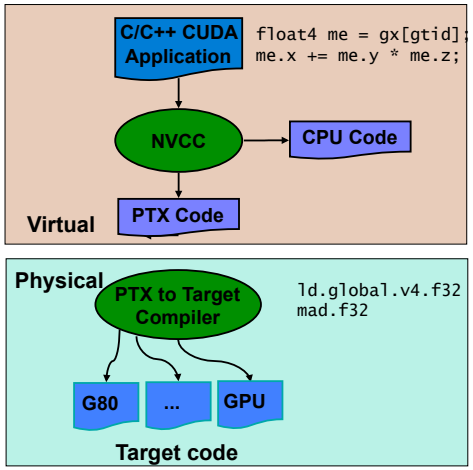
GPU is viewed as a compute device operating as a coprocessor to the main CPU (host)

- Data-parallel, compute intensive functions should be off-loaded to the device
- Functions that are executed many times, but independently on different data, are prime candidates
  - I.e. body of `for`-loops
- A function compiled for the device is called a *kernel*
- The kernel is executed on the device as many different *threads*
- Both host (CPU) and device (GPU) manage their own memory, *host memory* and *device memory*
  - Data can be copied between them

CUDA Software Development Kit



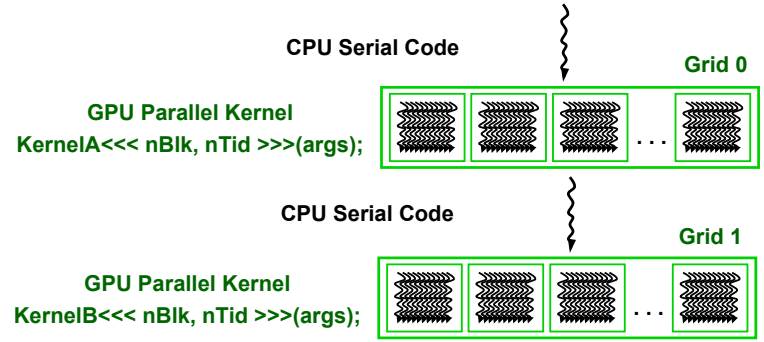
Compiling a CUDA Program



- Parallel Thread eXecution (PTX)
  - Virtual Machine and ISA
  - Programming model
  - Execution resources and state

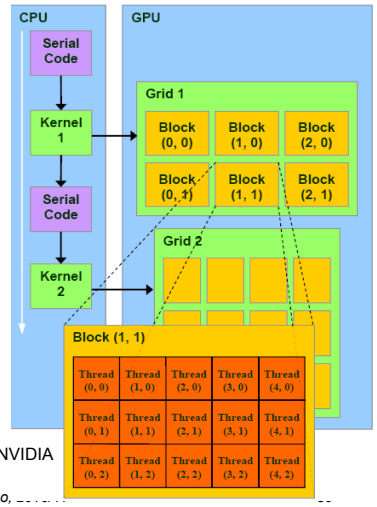
CUDA basic model: Single-Program Multiple-Data (SPMD)

- CUDA integrated CPU + GPU application C program
  - Serial C code executes on CPU
  - Parallel Kernel C code executes on GPU thread blocks



Programming Model: SPMD + SIMT/SIMD

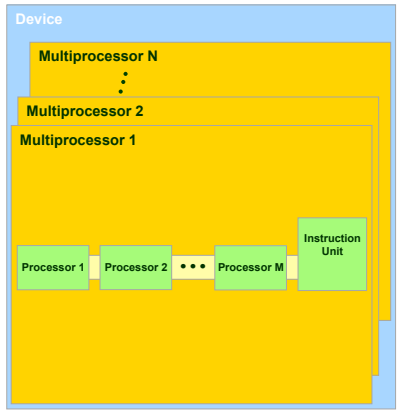
- Hierarchy
  - Device => Grids
  - Grid => Blocks
  - Block => Warps
  - Warp => Threads
- Single kernel runs on multiple blocks (SPMD)
- Threads within a warp are executed in a lock-step way called single-instruction multiple-thread (SIMT)
- Single instruction are executed on multiple threads (SIMD)
  - Warp size defines SIMD granularity (G80 : 32 threads)
- Synchronization within a block using shared memory



Courtesy NVIDIA

Hardware Implementation : a set of SIMD Processors

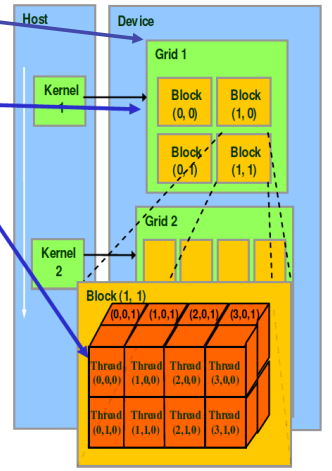
- Device
  - a set of multiprocessors
- Multiprocessor
  - a set of 32-bit SIMD processors



Courtesy NVIDIA

The Computational Grid: Block IDs and Thread IDs

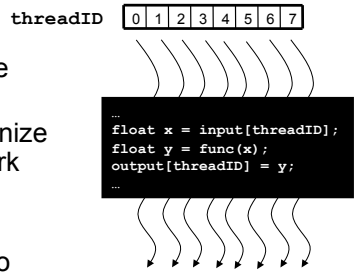
- A kernel runs on a computational grid of thread blocks
  - Threads share global memory
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- A thread block is a batch of threads that can cooperate by:
  - Sync their execution w/ barrier
  - Efficiently sharing data through a low latency shared memory
  - Two threads from two different blocks cannot cooperate



CUDA Thread Block

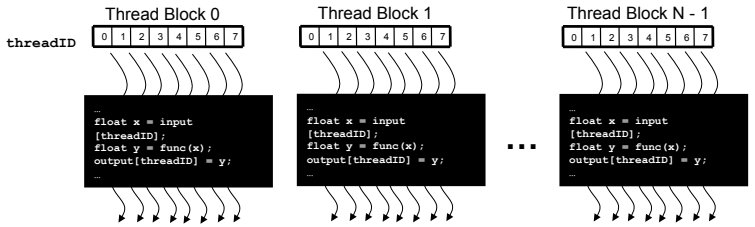
- Programmer declares (Thread) Block:
  - Block size 1 to **512** concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- All threads in a Block execute the same thread program
- Threads share data and synchronize while doing their share of the work
- Threads have thread id numbers within Block
- Thread program uses thread id to select work and address shared data

CUDA Thread Block



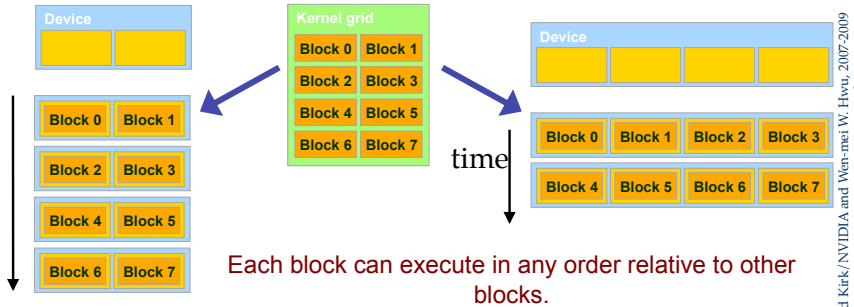
Thread Blocks: Scalable Cooperation

- Divide monolithic thread array into multiple blocks
  - Threads within a block cooperate via **shared memory, atomic operations** and **barrier synchronization**
  - Threads in different blocks cannot cooperate



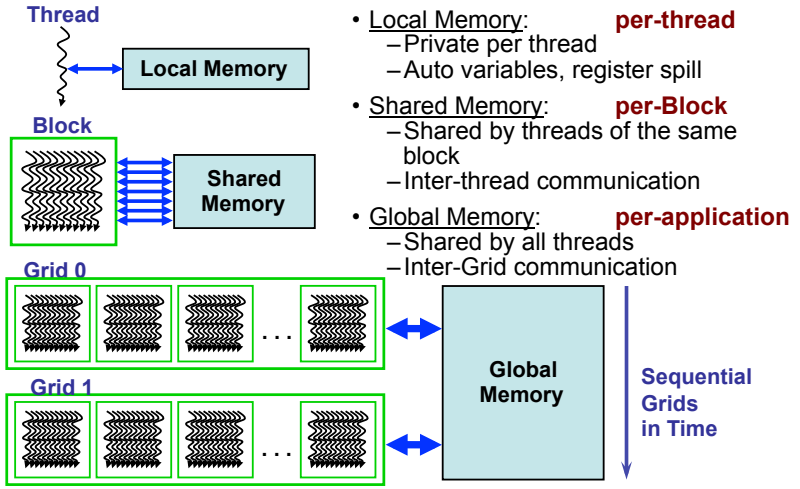
Transparent Scalability

- Hardware is free to assigns blocks to any processor at any time
  - A kernel scales across any number of parallel processors



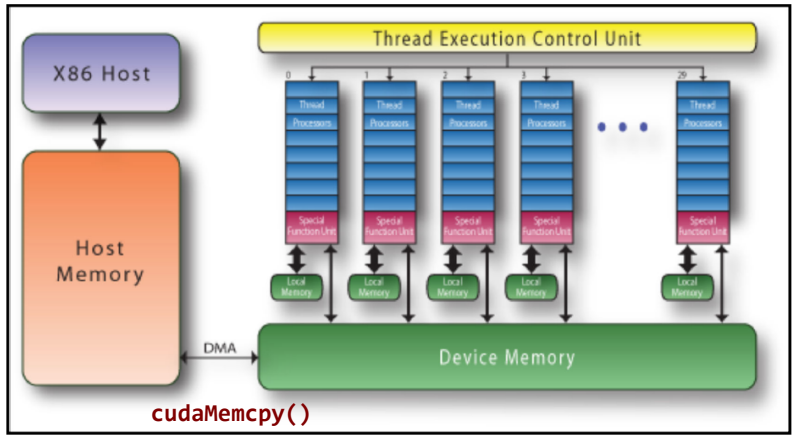
Each block can execute in any order relative to other blocks.

Parallel Memory Sharing



- Local Memory: **per-thread**
  - Private per thread
  - Auto variables, register spill
- Shared Memory: **per-Block**
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: **per-application**
  - Shared by all threads
  - Inter-Grid communication

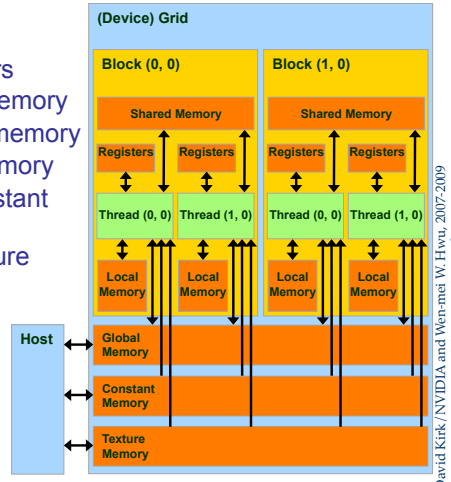
Memory model



NVIDIA GPU Accelerator Block Diagram

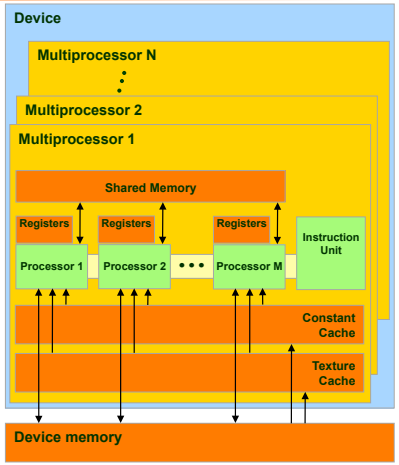
CUDA Memory Model Overview (1)

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



Hardware Implementation: Memory Architecture

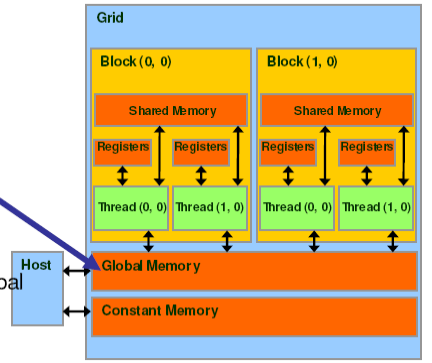
- Device memory (DRAM)
  - Slow (2~300 cycles)
  - Local, global, constant, and texture memory
- On-chip memory
  - Fast (1 cycle)
  - Registers, shared memory, constant/texture cache



Courtesy NVIDIA

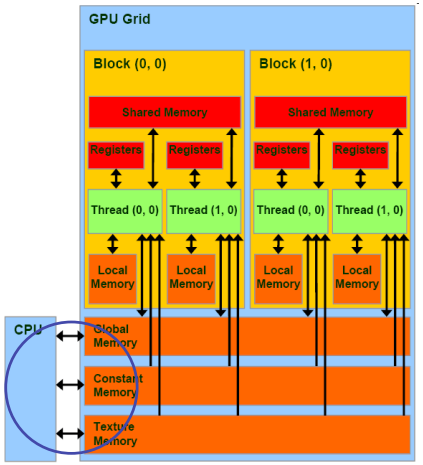
CUDA Memory Model Overview (2)

- cudaMalloc()
  - Allocates object in the device Global Memory
  - Two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** of allocated object
- cudaFree()
  - Frees object from device Global Memory
    - **Pointer** to freed object



CUDA Memory Model Overview (3)

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
- Transfer is asynchronous



CUDA Host-Device Data Transfer

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToHost are symbolic constants

cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);

CUDA: Features available on GPU

- Double and single precision
- Standard mathematical functions
  - `sinf`, `powf`, `atanf`, `ceil`, `min`, `sqrtf`, etc.
- Atomic memory operations
  - `atomicAdd`, `atomicMin`, `atomicAnd`, `atomicCAS`, etc.
- These work on both global and shared memory

CUDA: Minimal extensions to C/C++

- Declaration specifiers to indicate where things live

```
__global__ void KernelFunc(...);   // kernel callable from host
__device__ void DeviceFunc(...);   // function callable on device
__device__ int  GlobalVar;         // variable in device memory
__shared__ int  SharedVar;         // in per-block shared memory
```

- Extend function invocation syntax for parallel kernel launch

```
KernelFunc<<<500, 128>>>(...);     // 500 blocks, 128 threads each
```

- Special variables for thread identification in kernels

```
dim3 threadIdx;  dim3 blockIdx;  dim3 blockDim;
```

- Intrinsics that expose specific operations in kernel code

```
__syncthreads();                   // barrier synchronization
```

CUDA: Runtime support

- Explicit memory allocation returns pointers to GPU memory
  - `cudaMalloc()`, `cudaFree()`
- Explicit memory copy for host ↔ device, device ↔ device
  - `cudaMemcpy()`, `cudaMemcpy2D()`, ...
- Texture management
  - `cudaBindTexture()`, `cudaBindTextureToArray()`, ...
- OpenGL & DirectX interoperability
  - `cudaGLMapBufferObject()`, `cudaD3D9MapVertexBuffer()`, ...

Hello World: Vector Addition

```
//Compute vector sum C=A+B
//Each thread performs one pairwise addition
__global__ void vecAdd(float* a, float* b, float* c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main() {
    //Run N/256 blocks of 256 threads each
    vecAdd<<<N/256, 256>>>(d_a, d_b, d_c);
}
```


Example: Vector Addition Host Code

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

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

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

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

Courtesy NVIDIA

Example – Elementwise Matrix Addition

CPU Program

```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

Example – Elementwise Matrix Addition

CPU Program

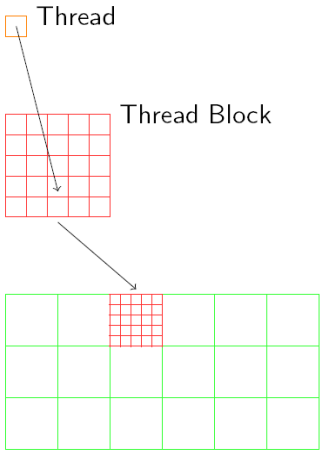
```
void add_matrix
  ( float* a, float* b, float* c, int N ) {
  int index;
  for ( int i = 0; i < N; ++i )
    for ( int j = 0; j < N; ++j ) {
      index = i + j*N;
      c[index] = a[index] + b[index];
    }
}

int main() {
  add_matrix( a, b, c, N );
}
```

CUDA Program

```
__global__ add_matrix
  ( float* a, float* b, float* c, int N ) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}

int main() {
  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( a, b, c, N );
}
```

The nested `for`-loops are replaced with an implicit grid

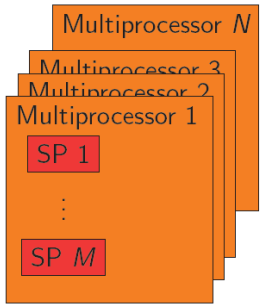
Recap



Multiple levels of parallelism

- Thread block
  - Up to 512 threads per block
  - Communicate via shared memory
  - Threads guaranteed to be resident
  - `threadIdx, blockIdx`
  - `__syncthreads()`
- Grid of thread blocks
  - `f<<<N, T>>>( a, b, c)`
  - Communicate via global memory

How threads are executed



- A GPU consist of N multiprocessors (MP)
- Each MP has M scalar processors (SP)
- Each MP processes batches of blocks
  - A block is processed by only one MP
- Each block is split into SIMD groups of threads called warps
  - A warp is executed physically in parallel
- A scheduler switches between warps
- A warp contains threads of consecutive, increasing thread ID
- The warp size is 32 threads today

Know the arithmetic cost of operations

- 4 clock cycles:
  - Floating point: add, multiply, fused multiply-add
  - Integer add, bitwise operations, compare, min, max
- 16 clock cycles:
  - reciprocal, reciprocal square root, `__log(x)`, 32-bit integer multiplication
- 32 clock cycles:
  - `__sin(x)`, `__cos(x)` and `__exp(x)`
- 36 clock cycles:
  - Floating point division (24-bit version in 20 cycles)
- Particularly costly:
  - Integer division, modulo
  - Remedy: Replace with shifting whenever possible
- Double precision (when available) will perform at half the speed

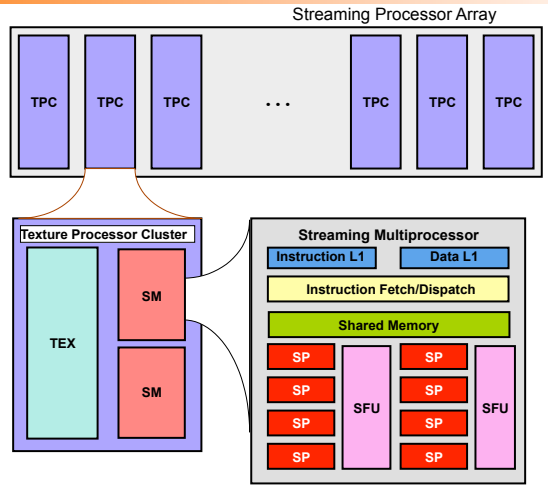
Mapping CUDA to Nvidia GPUs

- CUDA is designed to be functionally forgiving
  - First priority: make things work. Second: get performance.
- However, to get good performance, one must understand how CUDA is mapped to NVidia GPUs
- Threads:
  - each thread is a SIMD vector lane
- Warps:
  - A SIMD instruction acts on a “warp”
  - Warp width is 32 elements: ***LOGICAL*** SIMD width
- Thread blocks:
  - Each thread block is scheduled onto a processor
  - Peak efficiency requires multiple thread blocks per processor

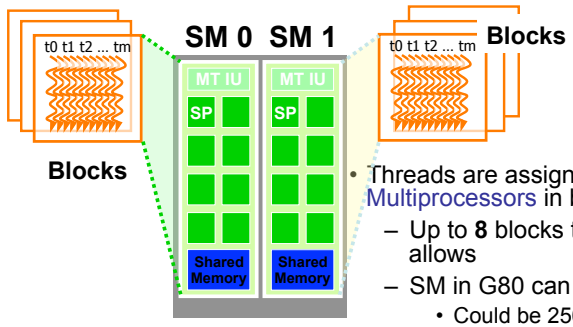
Mapping CUDA to a GPU, *cont.*

- The GPU is very deeply pipelined
  - Throughput machine, trying to hide memory latency
- This means that performance depends on the number of thread blocks which can be allocated on a processor
- Therefore, resource usage costs performance:
  - More registers => Fewer thread blocks
  - More shared memory usage => Fewer thread blocks
- It is often worth trying to reduce register count in order to get more thread blocks to fit on the chip
  - For previous architectures, 10 registers or less per thread meant full occupancy
  - For GTX280, target 16 registers or less per thread

GeForce-8 Series HW Overview



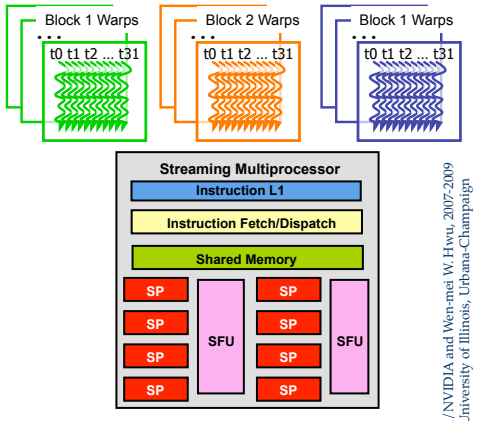
G80 Example: Executing Thread Blocks



- Threads are assigned to Streaming Multiprocessors in block granularity
  - Up to **8** blocks to each SM as resource allows
  - SM in G80 can take up to **768** threads
    - Could be 256 (threads/block) * 3 blocks
    - Or 128 (threads/block) * 6 blocks, etc.
- Threads run concurrently
  - SM maintains thread/block id #s
  - SM manages/schedules thread exec

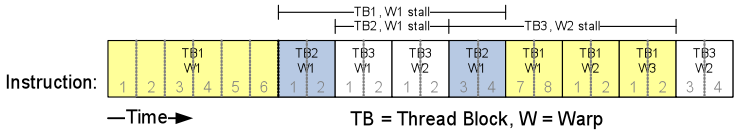
G80 Example: Thread Scheduling

- Each Block is executed as 32-thread Warps
  - An implementation decision, not part of the CUDA programming model
  - Warps are scheduling units in SM
- If 3 blocks are assigned to an SM and each block has 256 threads, how many Warps are there in an SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps

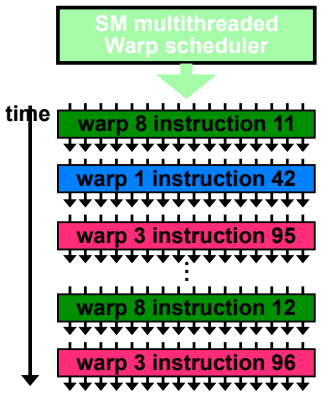


G80 Example: Scoreboarding

- SM implements zero-overhead warp scheduling
  - At any time, only one of the warps is executed by SM
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible warps are selected for execution on a prioritized scheduling policy
  - All threads in a warp execute the same instruction when selected



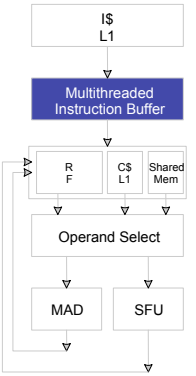
SM Warp Scheduling



- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80
  - If one global memory access is needed for every 4 instructions
  - A minimal of 13 Warps are needed to fully tolerate 200-cycle memory latency

SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - from instruction L1 cache
  - into any instruction buffer slot
- Issue one “ready-to-go” warp instruction/cycle
  - from any warp - instruction buffer slot
  - operand scoreboarding used to prevent hazards
- Issue selection based on round-robin/age of warp
- SM broadcasts the same instruction to 32 Threads of a Warp



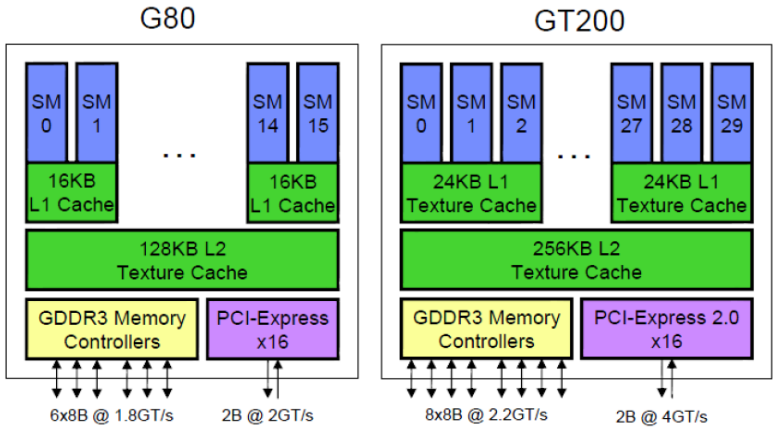
G80 Block Granularity Considerations

- For Matrix Multiplication using multiple blocks, should I use 8X8, 16X16 or 32X32 blocks?
  - For 8X8, we have 64 threads per Block. Since each SM can take up to 768 threads, there are 12 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!
  - For 16X16, we have 256 threads per Block. Since each SM can take up to 768 threads, it can take up to 3 Blocks and achieve full capacity unless other resource considerations overrule.
  - For 32X32, we have 1024 threads per Block. Not even one can fit into an SM!

Famílias de GPU da NVidia

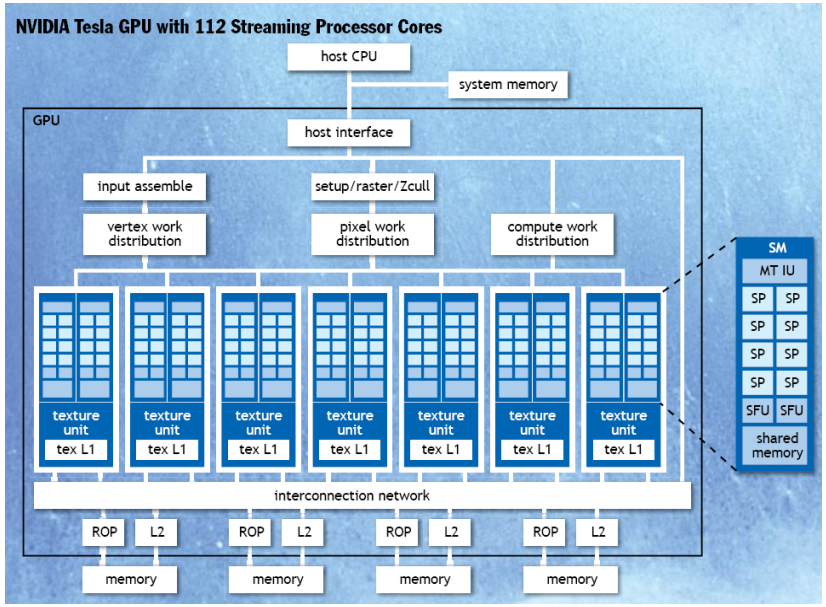
GPU	G80	GT200	Fermi
Transistors	681 million	1.4 billion	3.0 billion
CUDA Cores	128	240	512
Double-Precision Floating Point	None	30 FMA ops per clock	256 FMA ops per clock
Single-Precision Floating Point	128 MADD ops per clock	240 MADD ops per clock	512 FMA ops per clock
Warp Schedulers per Streaming Multiprocessor (SM)	1	1	2
Special Function Units per SM	2	2	4
Shared Memory per SM	16KB	16KB	Configurable 48KB or 16KB
L1 Cache per SM	None	None	Configurable 16KB or 48KB
L2 Cache	None	None	768KB
ECC Memory Protection	No	No	Yes
Concurrent Kernels	No	No	Up to 16

G80 & GT200

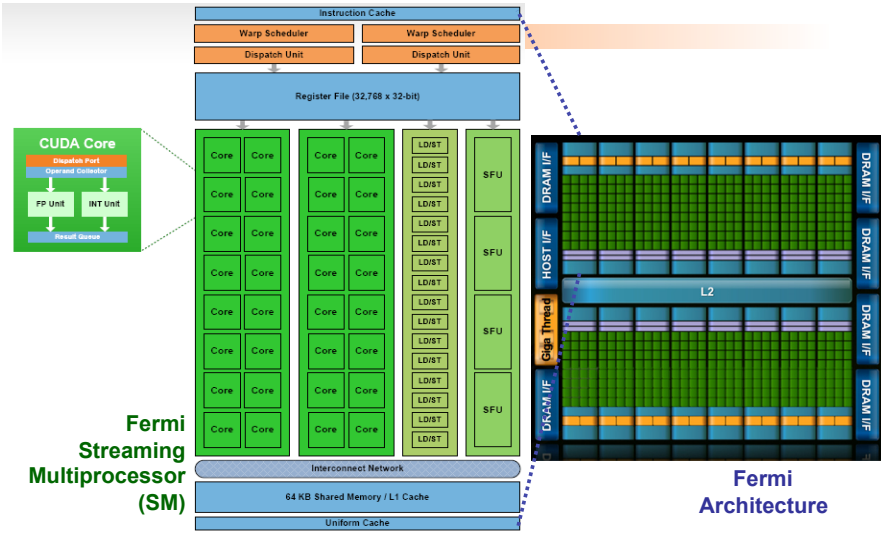


Texture/Processor Cluster, TPC: G80 and G200

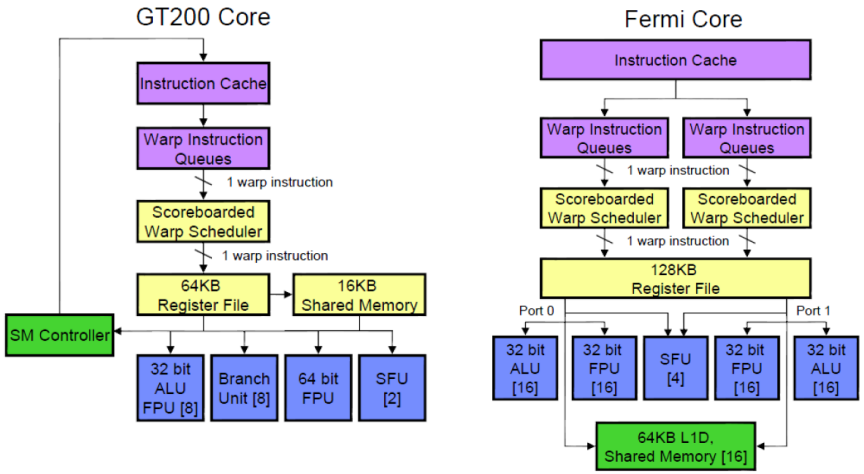
G80 **GT200**



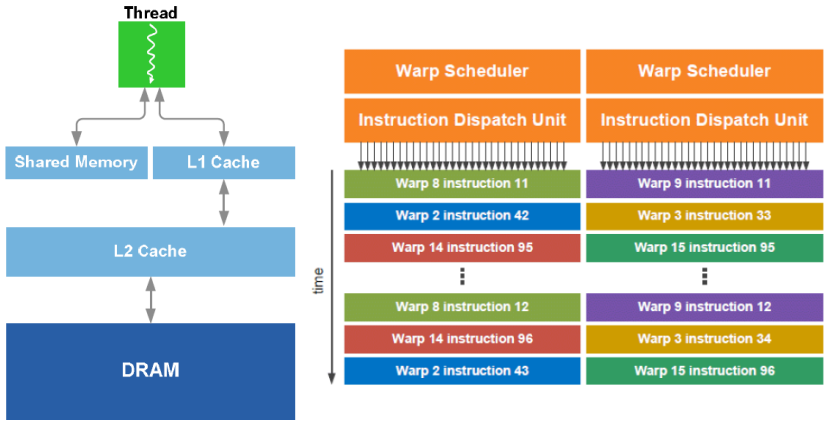
The new NVidia Fermi architecture



GT200 and Fermi cores



Fermi: Multithreading and Memory Hierarchy



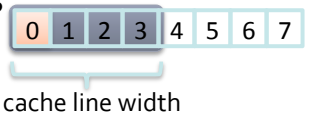
Memory, Memory, Memory

- A many core processor ≡ A device for turning a compute bound problem into a memory bound problem

- Lots of processors, only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

- Virtually all processors have SIMD memory subsystems



- This has two effects:
  - Sparse access wastes bandwidth

2 words used, 8 words loaded: ¼ effective bandwidth

  - Unaligned access wastes bandwidth

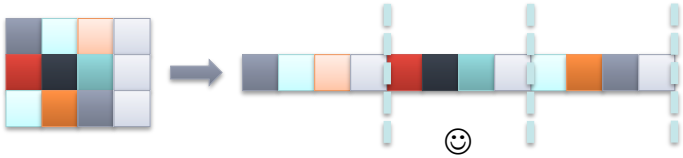
4 words used, 8 words loaded: ½ effective bandwidth

Coalescing

- Current GPUs don't have cache lines as such, but they do have similar issues with alignment and sparsity
- NVidia GPUs have a "coalescer", which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should:
  - Present a set of unit strided loads (dense accesses)
  - Keep sets of loads aligned to vector boundaries

Data Structure Padding

- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern



Experiences with CUDA

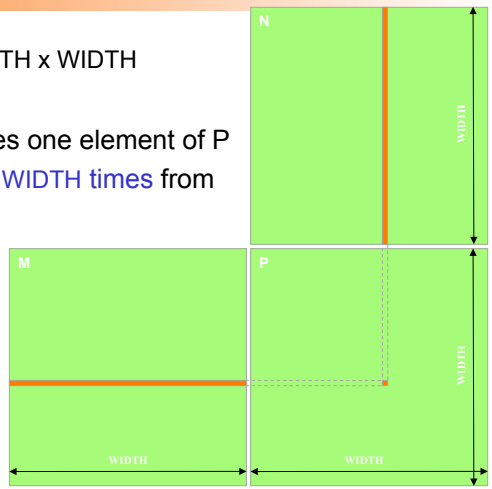
- Matrix multiplication:
  design step-by-step
- Matrix addition:
  code analysis step-by-step

A Simple Running Example
Matrix Multiplication

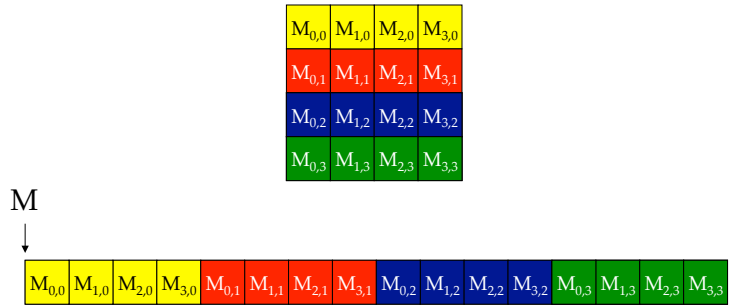
- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - Local, register usage
  - Thread ID usage
  - Memory data transfer API between host and device
  - Assume square matrix for simplicity

Programming Model:
Square Matrix Multiplication Example

- P = M * N of size WIDTH x WIDTH
- Without tiling:
  - One thread calculates one element of P
  - M and N are loaded WIDTH times from global memory

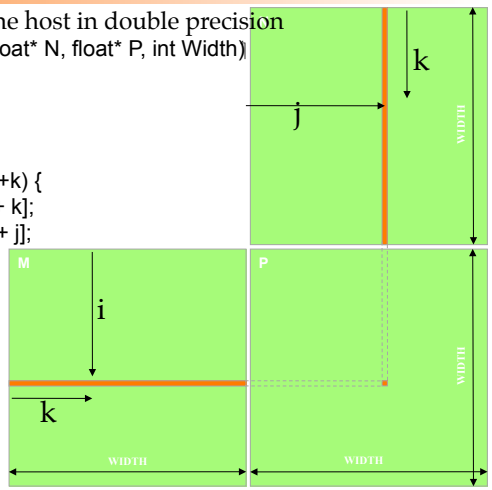


Memory Layout of a Matrix in C



Step 1: Matrix Multiplication
A Simple Host Version in C

```
// Matrix multiplication on the host in double precision
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)
        for (int j = 0; j < Width; ++j) {
            double sum = 0;
            for (int k = 0; k < Width; ++k) {
                double a = M[i * width + k];
                double b = N[k * width + j];
                sum += a * b;
            }
            P[i * Width + j] = sum;
        }
}
```



Step 2: Input Matrix Data Transfer
(Host-side Code)

```
void MatrixMulOnDevice(float* M, float* N, float* P, int Width)
{
    int size = Width * Width * sizeof(float);
    float* Md, Nd, Pd;
    …

1.  // Allocate and Load M, N to device memory
    cudaMalloc(&Md, size);
    cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

    cudaMalloc(&Nd, size);
    cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

    // Allocate P on the device
    cudaMalloc(&Pd, size);
```


Step 3: Output Matrix Data Transfer
(Host-side Code)

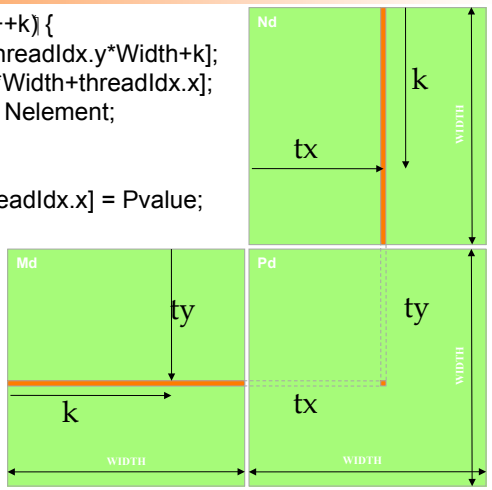
```
2.  // Kernel invocation code – to be shown later
    …

3.   // Read P from the device
     cudaMemcpy(P, Pd, size,
     cudaMemcpyDeviceToHost);

     // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);
    }
```


Step 4: Kernel Function

```
// Matrix multiplication kernel – per thread code

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd,
int Width)
{

    // Pvalue is used to store the element of the matrix
    // that is computed by the thread
    float Pvalue = 0;
```

Step 4: Kernel Function (cont.)

```
  for (int k = 0; k < Width; ++k) {
      float Melement = Md[threadIdx.y*Width+k];
      float Nelement = Nd[k*Width+threadIdx.x];
      Pvalue += Melement * Nelement;
  }

  Pd[threadIdx.y*Width+threadIdx.x] = Pvalue;
}
```

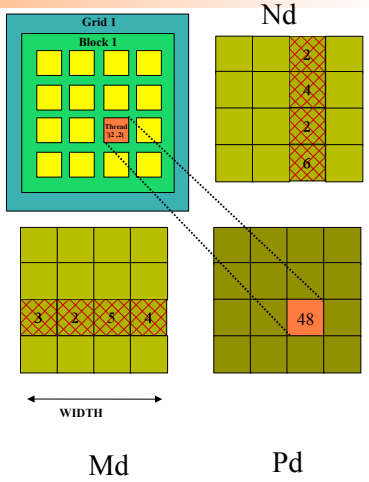


Step 5: Kernel Invocation (Host-side Code)

```
// Setup the execution configuration
  dim3 dimGrid(1, 1);
 dim3 dimBlock(Width, Width);


// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```


Only One Thread Block Used

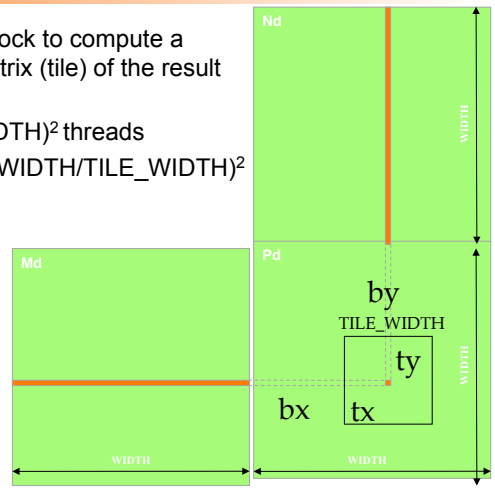
- One Block of threads compute matrix Pd
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
  - Compute to off-chip memory access ratio close to 1:1 (not very high)
- Size of matrix limited by the number of threads allowed in a thread block



Step 7: Handling Arbitrary Sized Square Matrices

- Have each 2D thread block to compute a $(TILE\_WIDTH)^2$ sub-matrix (tile) of the result matrix
  - Each has $(TILE\_WIDTH)^2$ threads
- Generate a 2D Grid of $(WIDTH/TILE\_WIDTH)^2$ blocks

You still need to put a loop around the kernel call for cases where WIDTH/TILE_WIDTH is greater than max grid size (64K)!



Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

Compileable example

Set grid size

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

Compileable example

Compute kernel

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

Compileable example

CPU Mem Allocation

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

GPU Mem Allocation

Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Copy data to GPU

Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Execute kernel

Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd, size, cudaMemcpyDeviceToHost );

  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```

Copy result back to CPU

Compileable example

```
const int N = 1024;
const int blocksize = 16;

__global__
void add_matrix( float* a, float *b, float *c, int N )
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  int index = i + j*N;
  if ( i < N && j < N )
    c[index] = a[index] + b[index];
}
```

- Store in source file (i.e. MatrixAdd.cu)
- Compile with nvcc MatrixAdd.cu
- Run
- Enjoy the benefits of parallelism!

```
int main() {
  float *a = new float[N*N];
  float *b = new float[N*N];
  float *c = new float[N*N];

  for ( int i = 0; i < N*N; ++i ) {
    a[i] = 1.0f; b[i] = 3.5f; }

  float *ad, *bd, *cd;
  const int size = N*N*sizeof(float);
  cudaMalloc( (void**)&ad, size );
  cudaMalloc( (void**)&bd, size );
  cudaMalloc( (void**)&cd, size );

  cudaMemcpy( ad, a, size, cudaMemcpyHostToDevice );
  cudaMemcpy( bd, b, size, cudaMemcpyHostToDevice );

  dim3 dimBlock( blocksize, blocksize );
  dim3 dimGrid( N/dimBlock.x, N/dimBlock.y );
  add_matrix<<<dimGrid, dimBlock>>>( ad, bd, cd, N );

  cudaMemcpy( c, cd,
```

Clean up and return

```
  cudaFree( ad ); cudaFree( bd ); cudaFree( cd );
  delete[] a; delete[] b; delete[] c;

  return EXIT_SUCCESS;
}
```


Summary- Typical Structure of a CUDA Program

- Global variables declaration
  - __host__
  - __device__... __global__, __constant__, __texture__
- Function prototypes
  - __global__ void kernelOne(…)
  - float handyFunction(…)
- Main ()
  - allocate memory space on the device – cudaMalloc(&d_GlblVarPtr, bytes)
  - transfer data from host to device – cudaMemCpy(d_GlblVarPtr, h_Gl…)
  - execution configuration setup
  - kernel call – kernelOne<<<execution configuration>>>(args…);
  - transfer results from device to host – cudaMemCpy(h_GlblVarPtr,…)
  - optional: compare against golden (host computed) solution

  repeat as needed
- Kernel – void kernelOne(type args,…)
  - variables declaration - __local__, __shared__
    - automatic variables transparently assigned to registers or local memory
  - syncthreads()…
- Other functions
  - float handyFunction(int inVar…);

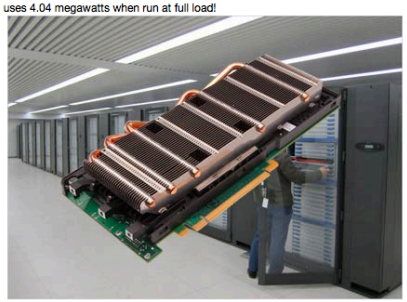
CUDA Summary

- CUDA is a SPMD+SIMD programming model for manycore processors
- It abstracts SIMD, making it easy to use wide SIMD vectors
- It provides good performance on today's GPUs
- In the near future, CUDA-like approaches will map well to many processors & GPUs
- CUDA encourages SIMD friendly, highly scalable algorithm design and implementation

Latest News: #1 in TOP500 (Nov'2010)

NVIDIA Tesla M2050 GPUs Power World's Fastest Supercomputer - Tianhe-1A

Tianhe-1A, a new supercomputer revealed today at HPC 2010 China, has set a new performance record of 2.507 petaflops, as measured by the LINPACK benchmark, making it the fastest system in China and in the world today. The Tianhe-1A supercomputer uses 7,168 NVIDIA Tesla M2050 GPUs and 14,336 Intel Xeon CPUs. The 2.507 petaflop system uses 4.04 megawatts when run at full load!



28-Oct-10

"Tianhe-1A epitomizes modern heterogeneous computing by coupling massively parallel GPUs with multi-core CPUs, enabling significant achievements in performance, size and power. The system uses 7,168 NVIDIA Tesla M2050 GPUs and 14,336 CPUs; it would require more than 50,000 CPUs and twice as much floor space to deliver the same performance using CPUs alone. More importantly, a 2.507 petaflop system built entirely with CPUs would consume more than 12 megawatts. Thanks to the use of GPUs in a heterogeneous computing environment, Tianhe-1A consumes only 4.04 megawatts, making it 3 times more power efficient -- the difference in power consumption is enough to provide electricity to over 5000 homes for a year."