2110412 Parallel Comp Arch
CUDA: Parallel Programming on GPU

Natawut Nupairoj, Ph.D.
Department of Computer Engineering, Chulalongkorn University

Outline

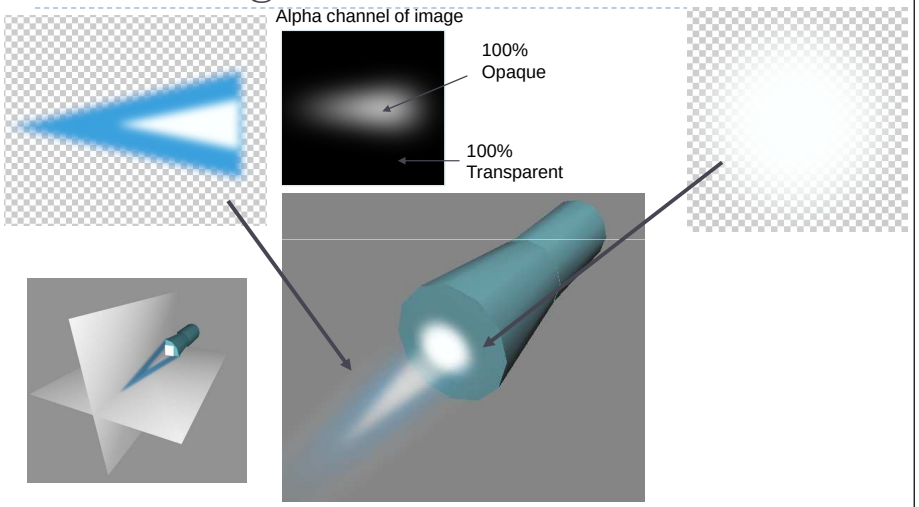
- Overview
- Introduction to CUDA
- CUDA Thread Model
- CUDA Memory Hierarchy and Memory Spaces
- CUDA Synchronization

Overview

- Modern graphics accelerators are called GPUs (Graphics Processing Units)
- 2 ways GPUs speed up graphics:
  - Pipelining: similar to pipelining in CPUs.
  - CPUs like Pentium 4 has 20 pipeline stages.
  - GPUs typically have 600-800 stages. -- very few branches & most of the functionality is fixed.

Source: Leigh, "Graphics Hardware Architecture & Miscellaneous Real Time Special Effects"

Rocket Engines



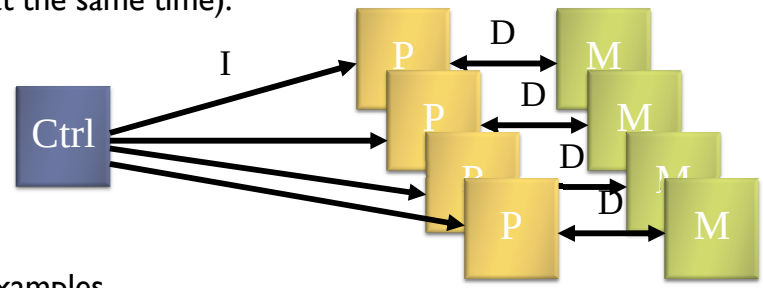
Overview

- Parallelizing
  - Process the data in parallel within the GPU. In essence multiple pipelines running in parallel.
  - Basic model is SIMD (Single Instruction Multiple Data) – ie same graphics algorithms but lots of polygons to process.

Source: Leigh,"Graphics Hardware Architecture & Miscellaneous Real Time Special Effects"

SIMD

- One control unit tells processing elements to compute (at the same time).



- Examples
  - TMC/CM-1, Maspar MP-1, Modern GPU

Modern GPU is More General Purpose – Lots of ALU's

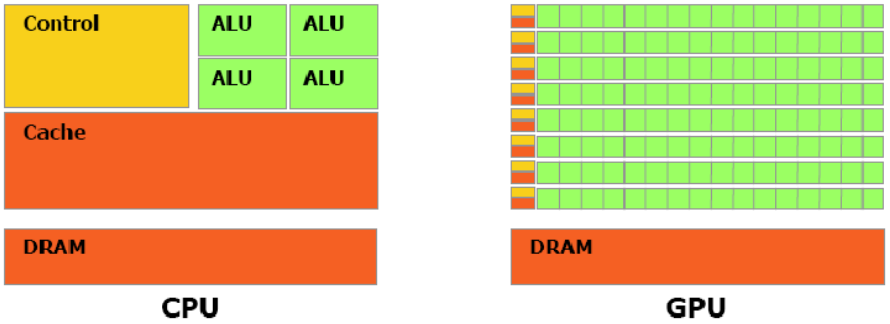
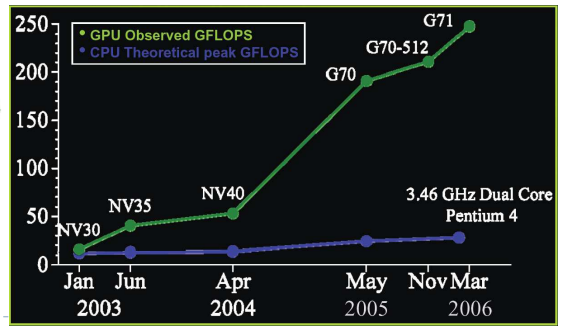


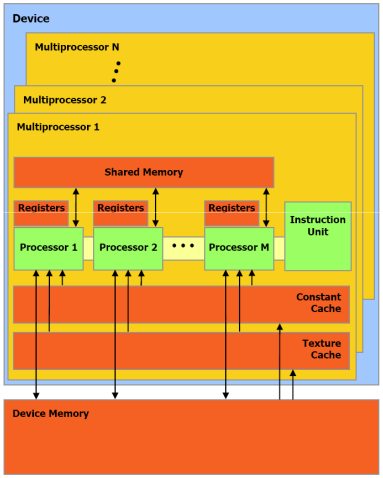
Figure 1-2. The GPU Devotes More Transistors to Data Processing

The nVidia G80 GPU

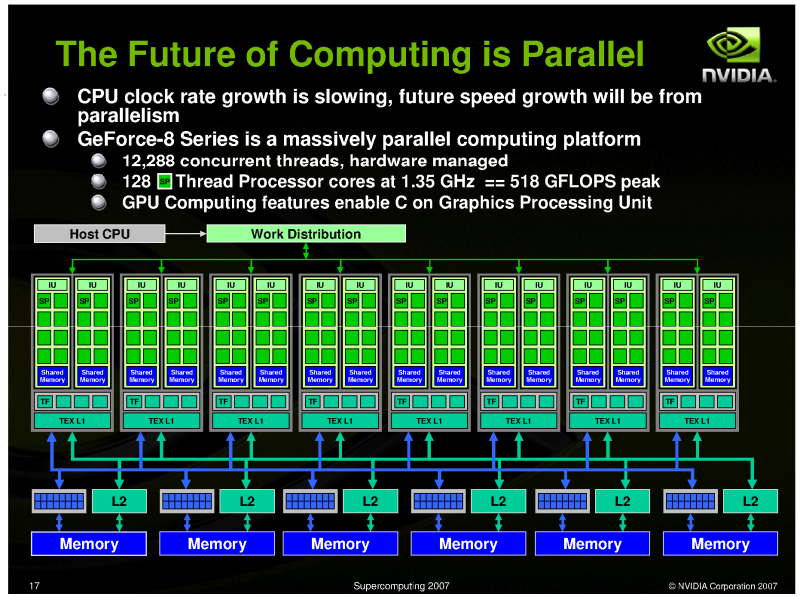
- **128 streaming floating point processors @1.5Ghz**
- **1.5 Gb Shared RAM with 86Gb/s bandwidth**
- **500 Gflop on one chip (single precision)**



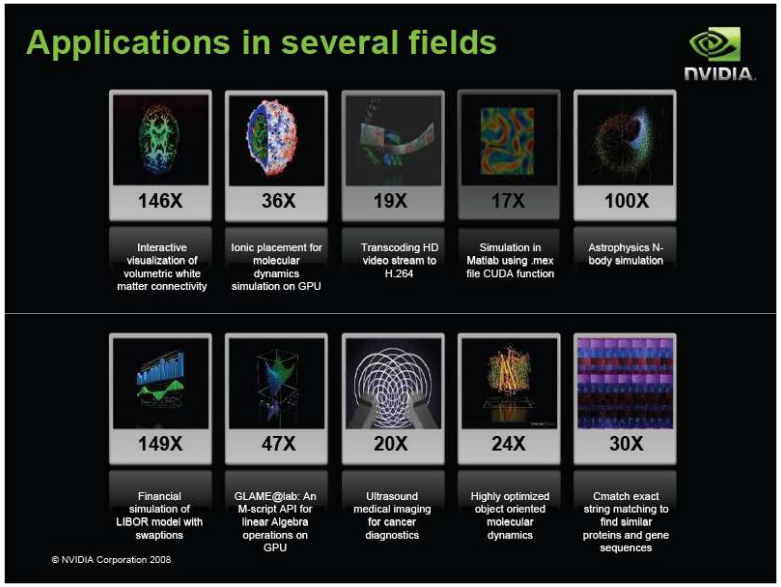
nVidia G80 GPU Architecture Overview



- •16 Multiprocessors Blocks
- •Each MP Block Has:
  - •8 Streaming Processors (IEEE 754 spfp compliant)
  - •16K Shared Memory
  - •64K Constant Cache
  - •8K Texture Cache
- •Each processor can access all of the memory at 86Gb/s, but with different latencies:
- •Shared – 2 cycle latency
- •Device – 300 cycle latency



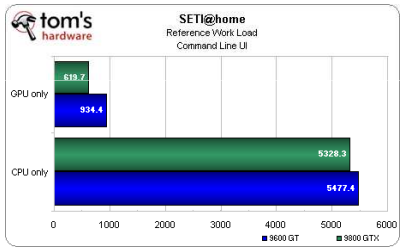
Source: Kirk, "Parallel Computing: What has changed lately?"



Source: CUDA Tutorial Workshop, ISC-2009

SETI@home and CUDA

- Run 5x to 10x times faster than CPU-only version



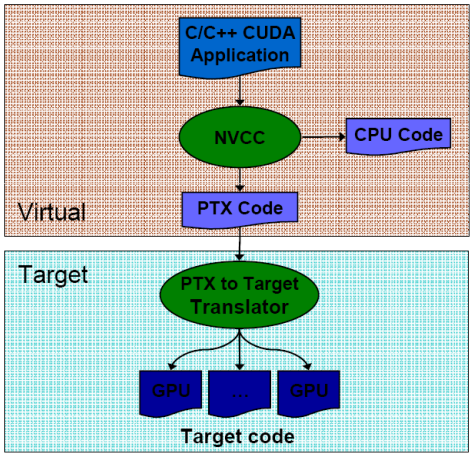
Introduction to CUDA

- nVidia introduced CUDA in November 2006
- Utilize parallel computing engine in GPU to solve complex computational problems
- CUDA is industry-standard C
  - Subset of C with extensions
  - Write a program for one thread
  - Instantiate it on many parallel threads
  - Familiar programming model and language
- CUDA is a scalable parallel programming model
  - Program runs on any number of processors without recompiling

CUDA Concept

- Co-Execution between Host (CPU) and Device (GPU)
- Parallel portions are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions
- Serial program with parallel kernels, all in C
  - Serial C code executes in a CPU thread
  - Parallel kernel C code executes in thread blocks across multiple processing elements

CUDA Development: nvcc

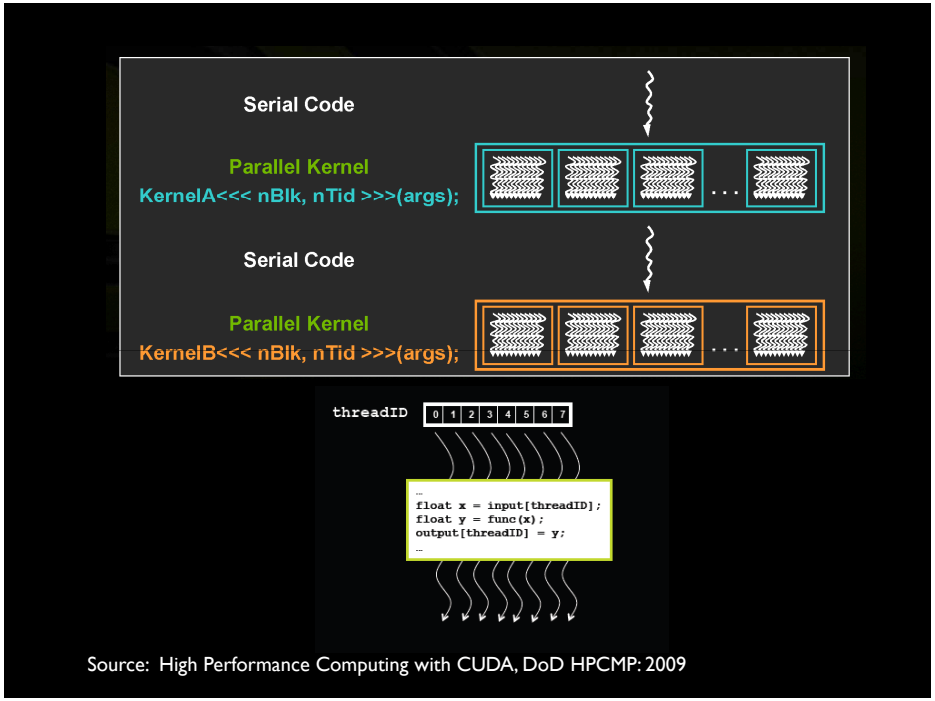


Normal C Program

```
void VecAdd_CPU(float* A, float* B, float* C, int N)
{
  for(int i=0 ; i < N ; i++)
       C[i] = A[i] + B[i];
}

void main()
{
  VecAdd_CPU(A, B, C, N);
}
```

CUDA Program

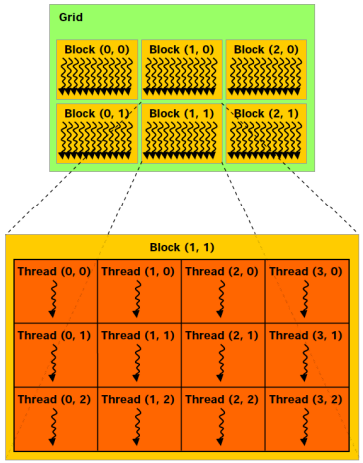
```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
  int i = threadIdx.x;
  C[i] = A[i] + B[i];
}

void main()
{
  // Kernel invocation
  VecAdd<<<1, N>>>(A, B, C);
}
```



Source: High Performance Computing with CUDA, DoD HPCMP: 2009

CUDA Thread Model

- CUDA Thread can be
  - one-dimensional
  - two-dimensional
  - three-dimensional
- Thread Hierarchy
  - Grid
  - (2-D) Block
  - (3-D) Thread



Calling CUDA Kernel

- Modified C function call syntax:

```
kernel<<<dim3 dG, dim3 dB>>>(…)
```

- Execution Configuration ("<<< >>>")
  - dG - dimension and size of grid in blocks
    - Two-dimensional: x and y
    - Blocks launched in the grid: dG.x*dG.y
  - dB - dimension and size of blocks in threads:
    - Three-dimensional: x, y, and z
    - Threads per block: dB.x*dB.y*dB.z
  - Unspecified dim3 fields initialize to 1

Example: Adding 2-D Matrix

```
// Kernel definition
__global__ void MatAdd(float A[M][N], float B[M][N], float C[M][N])
{
  int i = threadIdx.x;
  int j = threadIdx.y;
  C[i][j] = A[i][j] + B[i][j];
}
void main()
{
  // Kernel invocation
  dim3 dimBlock(M, N);
  MatAdd<<<1, dimBlock>>>(A, B, C);
}
```

CUDA Built-In Device Variables

- All __global__ and __device__ functions have access to these automatically defined variables
  - dim3 gridDim;
    - Dimensions of the grid in blocks (at most 2D)
  - dim3 blockDim;
    - Dimensions of the block in threads
  - dim3 blockIdx;
    - Block index within the grid
  - dim3 threadIdx;
    - Thread index within the block

Example: Adding 2-D Matrix

```
// Kernel definition
__global__ void MatAdd(float A[M][N], float B[M][N], float C[M][N])
{
  int i = blockIdx.x;
  int j = threadIdx.x;
  C[i][j] = A[i][j] + B[i][j];
}
void main()
{
  // Kernel invocation
  MatAdd<<<M, N>>>(A, B, C);
}
```

Example: Adding 2-D Matrix

```
// Kernel definition
__global__ void MatAdd(float A[M][N], float B[M][N], float C[M][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
       C[i][j] = A[i][j] + B[i][j];
}
int main()
{
  // Kernel invocation
  dim3 dimBlock(16, 16);
  dim3 dimGrid((M + dimBlock.x – 1) / dimBlock.x,
               (N + dimBlock.y – 1) / dimBlock.y);
  MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Function Qualifiers

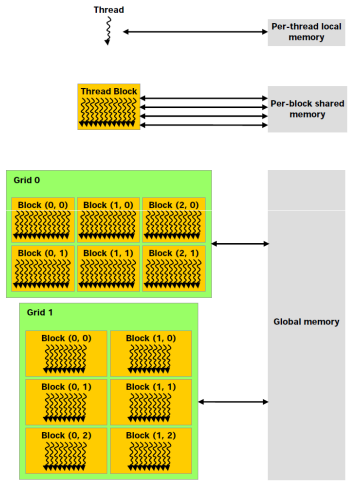
- Kernels designated by function qualifier:
  - __global__
    - Function called from host and executed on device
    - Must return void
- Other CUDA function qualifiers
  - __device__
    - Function called from device and run on device
    - Cannot be called from host code

Note on CUDA Kernel

- Kernels are C functions with some restrictions
  - Cannot access host memory
  - Must have void return type
  - No variable number of arguments ("varargs")
  - Not recursive
  - No static variables
- Function arguments automatically copied from host to device

CUDA Memory Hierarchy

- Each thread has private per-thread local memory
- All threads in a block have per-block shared memory
- All threads can access shared global memory



Exercise

```
int main()
{
  ...
  kernel<<<3, 5>>>( d_a );
  ...
}
```

Exercise

```
__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = 7;
}
```

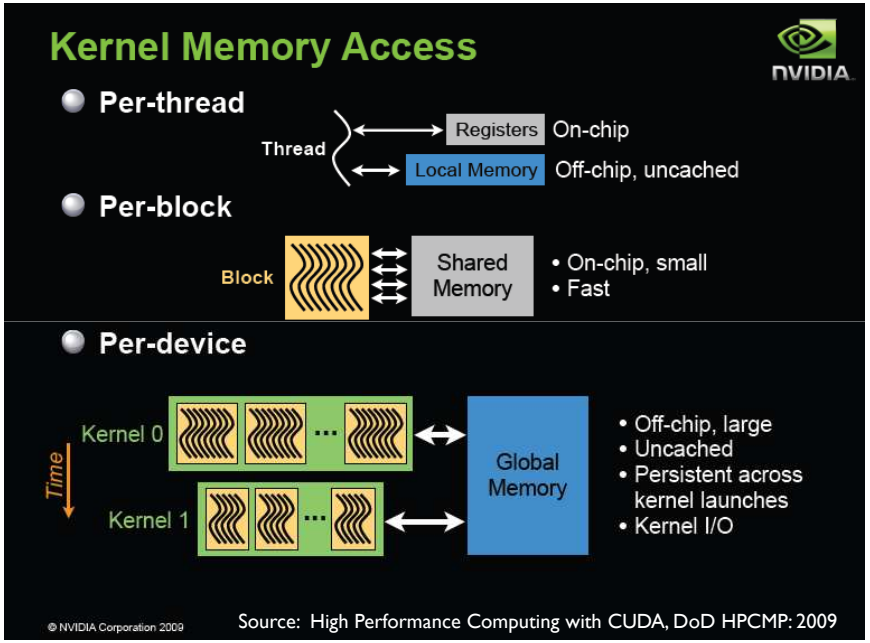
Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 2

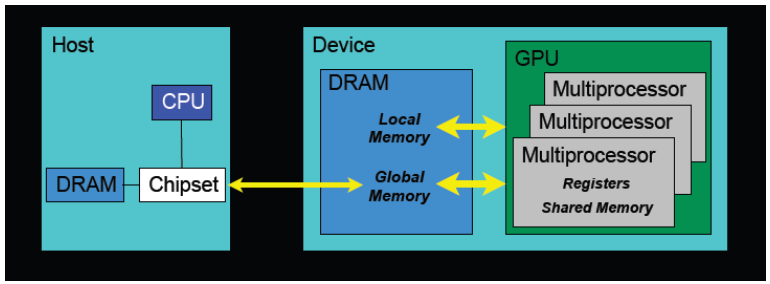
```
__global__ void kernel( int *a )
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 4 0 1 2 3 4 0 1 2 3 4 4



CUDA Host/Device Memory Spaces

- "Local" memory resides in device DRAM
  - Use registers and shared memory to minimize local memory use
- Host can read and write global memory but not shared memory



Source: High Performance Computing with CUDA, DoD HPCMP: 2009

Memory Spaces

- CPU and GPU have separate memory spaces
  - Data is moved across PCIe bus
  - Use functions to allocate/set/copy memory on GPU
    - Very similar to corresponding C functions
- Host (CPU) manages device (GPU) memory

```
cudaMalloc(void **pointer, size_t nbytes)
cudaMemset(void *pointer, int value, size_t count)
cudaFree(void *pointer)
```

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int *a_d = 0;
cudaMalloc( (void**)&a_d, nbytes );
cudaMemset( a_d, 0, nbytes);
cudaFree(a_d);
```

Host / Device Data Copies

```
cudaMemcpy(void *dst, void *src, size_t nbytes, enum
           cudaMemcpyKind direction);
```

- direction specifies locations (host or device) of src and dst
- Blocks CPU thread: returns after the copy is complete
- Doesn't start copying until previous CUDA calls complete
- enum cudaMemcpyKind
  - cudaMemcpyHostToDevice
  - cudaMemcpyDeviceToHost
  - cudaMemcpyDeviceToDevice


```
int main(void)
{
  float *a_h, *b_h; // host data
  float *a_d, *b_d; // device data
  int N = 14, nBytes, i ;

  nBytes = N*sizeof(float);
  a_h = (float *)malloc(nBytes);
  b_h = (float *)malloc(nBytes);
  cudaMalloc((void **) &a_d, nBytes);
  cudaMalloc((void **) &b_d, nBytes);

  for (i=0, i<N; i++) a_h[i] = 100.f + i;

  cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
  cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);
  cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);

  for (i=0; i< N; i++) assert( a_h[i] == b_h[i] );
  free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
  return 0;
}
```

Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- Memcopies are synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete
- Asynchronous CUDA calls provide:
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

Host Synchronization Example

```
…
// copy data from host to device
cudaMemcpy(a_d, a_h, numBytes, cudaMemcpyHostToDevice);
// execute the kernel
inc_gpu<<<ceil(N/(float)blocksize), blocksize>>>(a_d, N);
// run independent CPU code
run_cpu_stuff();
// copy data from device back to host
cudaMemcpy(a_h, a_d, numBytes, cudaMemcpyDeviceToHost);
…
```

GPU Thread Synchronization

- void __syncthreads();
- Synchronizes all threads in a block
  - Generates barrier synchronization instruction
  - No thread can pass this barrier until all threads in the block reach it
  - Used to avoid RAW / WAR / WAW hazards when accessing shared memory
- Allowed in conditional code only if the conditional is uniform across the entire thread block

CUDA Shared Memory

- __device__
  - Stored in global memory (large, high latency, no cache)
  - Allocated with cudaMalloc (__device__ qualifier implied)
  - Accessible by all threads
  - Lifetime: application
- __shared__
  - Stored in on-chip shared memory (very low latency)
  - Specified by execution configuration or at compile time
  - Accessible by all threads in the same thread block
  - Lifetime: thread block
- Unqualified variables:
  - Scalars and built-in vector types are stored in registers
  - Arrays may be in registers or local memory

Using Shared Memory

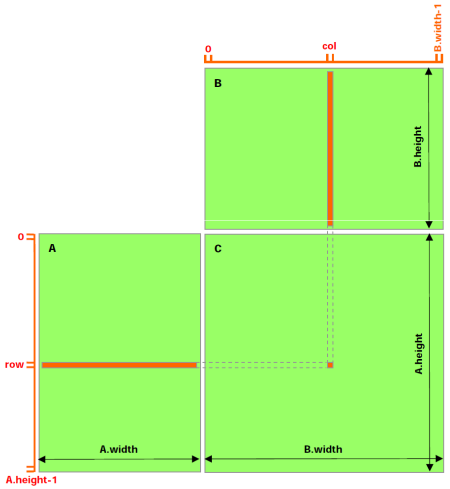
Size known at compile time

```
__global__ void kernel(…)
{
   …
   __shared__ float sData[256];
   …
}
int main(void)
{
   …
   kernel<<<nBlocks,blockSize>>>(…);
   …
}
```

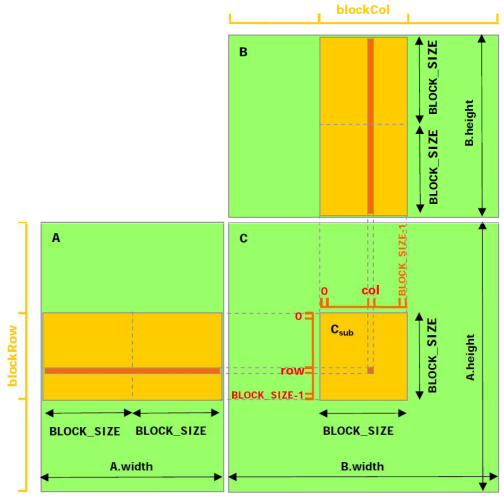
Size known at kernel launch

```
__global__ void kernel(…)
{
   …
   extern __shared__ float sData[];
   …
}
int main(void)
{
   …
   smBytes=blockSize*sizeof(float);
   kernel<<<nBlocks, blockSize,
         smBytes>>>(…);
…
}
```

Example: Matrix Multiplication version 1



Example: Matrix Multiplication version 2



Still A Specialized Processor

- Very Efficient For
  - Fast Parallel Floating Point Processing
  - Single Instruction Multiple Data Operations
  - High Computation per Memory Access
- Not As Efficient For
  - Double Precision (need to test performance)
  - Logical Operations on Integer Data
  - Branching-Intensive Operations
  - Random Access, Memory-Intensive Operations

How to Build CUDA on Windows XP

- Requirements for building CUDA program
  - CUDA software (available at no cost from http://www.nvidia.com/cuda)
    - CUDA toolkit
    - CUDA SDK
  - Microsoft Visual Studio 2005 or 2008, or the corresponding versions of Microsoft Visual C++ Express
  - CUDA VS Wizard (http://sourceforge.net/projects/cudavswizard/)
- Requirements for running CUDA
  - Using emulator in SDK (EmuDebug / EmuRelease)
  - CUDA-enabled GPU with device driver (version 185.xx+)
- See "CUDA Getting Started" for more details

Assignment

- Writing an CUDA program for Calculating PI
  - You must measure the elapsed time for calculation
- This is a team project
  - Each team can have 2-3 members
- Due date: 15 September 2009 at 18:00
- How to submit: sending email to "natawut.n@chula.ac.th"
- Note: I will use timestamp on your email