

2110412 Parallel Comp Arch

CUDA: Parallel Programming on GPU

Natawut Nupairoj, Ph.D.

Department of Computer Engineering, Chulalongkorn University

Outline

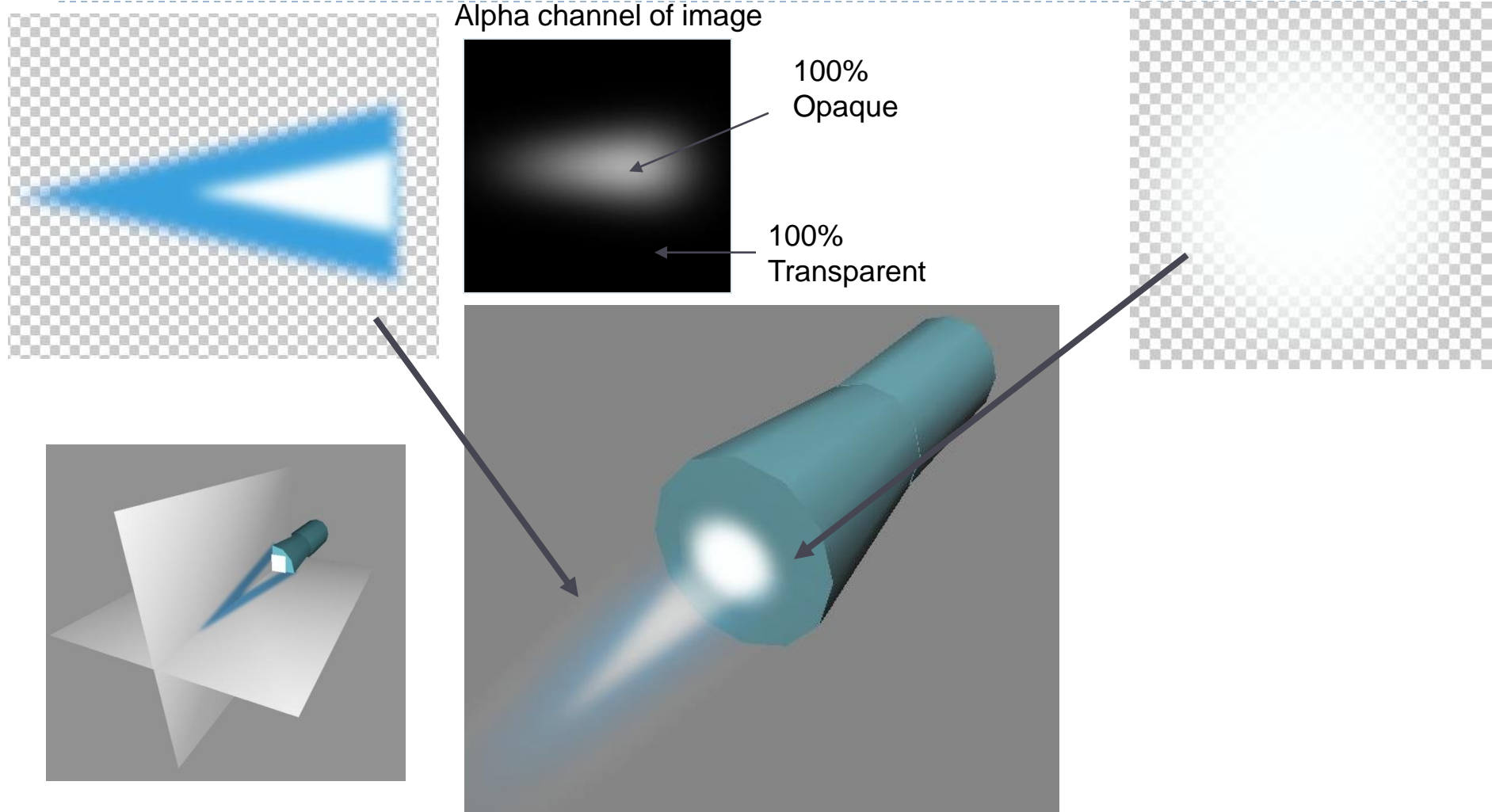
- ▶ Overview
- ▶ Parallel Computing with GPU
- ▶ Introduction to CUDA
- ▶ CUDA Thread Model
- ▶ CUDA Memory Hierarchy and Memory Spaces
- ▶ CUDA Synchronization



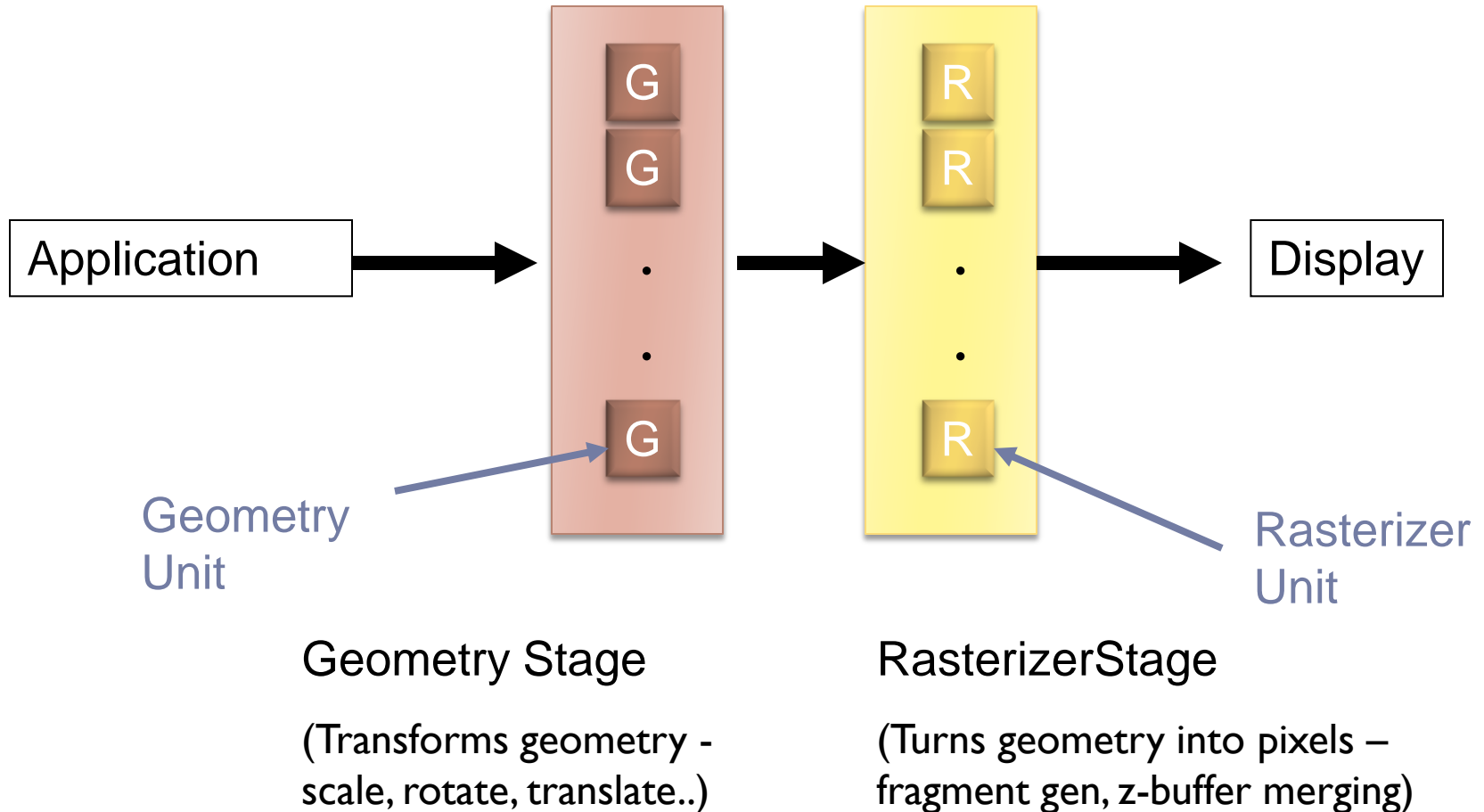
Overview

- ▶ 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.

Rocket Engines



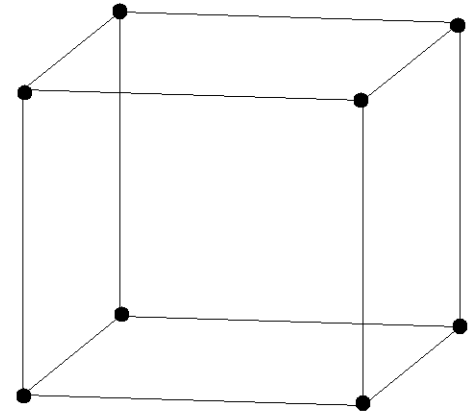
Typical Parallel Graphics Architecture



Transformation

- ▶ Performs a sequence of math operation on each vertex

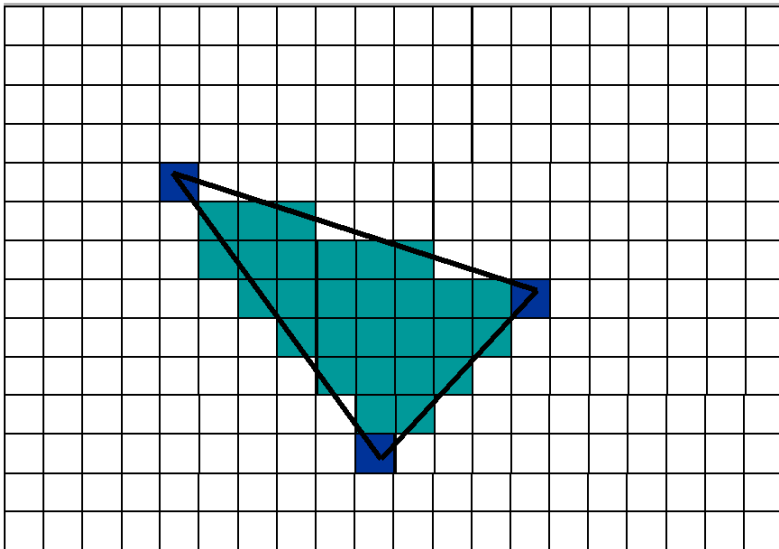
$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \times \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} \times \begin{bmatrix} * & * & * & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{bmatrix} = \begin{bmatrix} screen_x \\ screen_y \\ depth \\ 1 \end{bmatrix}$$



Rasterization

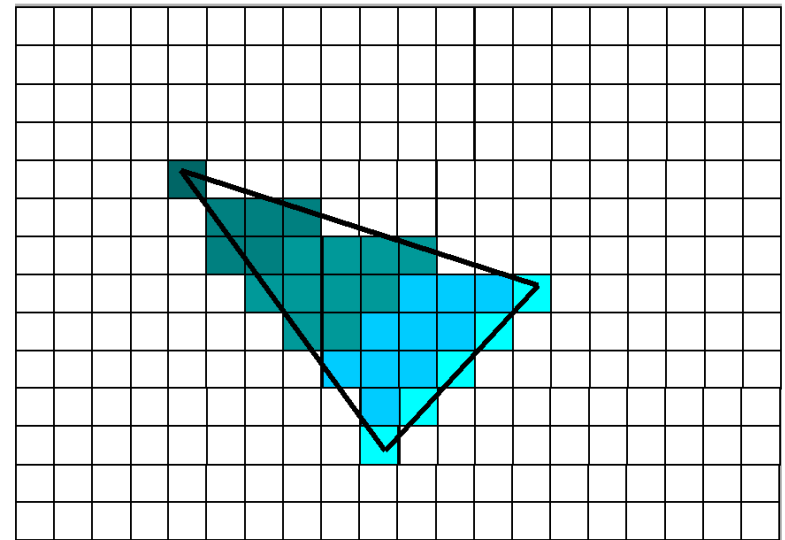
Rasterization

- ▶ Enumerates the pixels covered by triangles

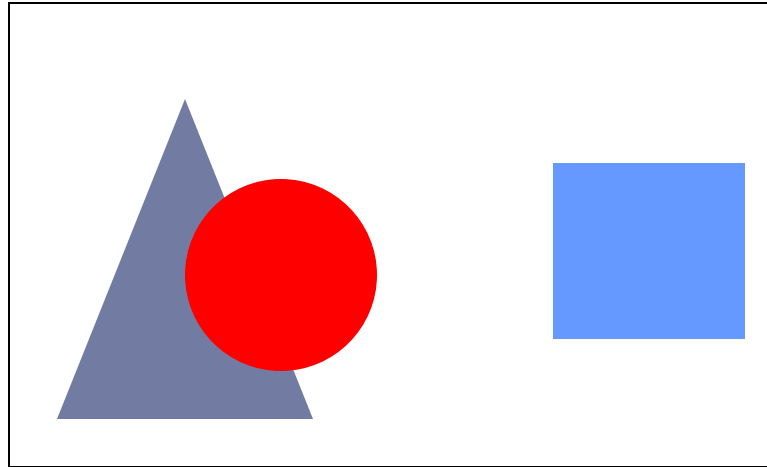


Fragment Processing

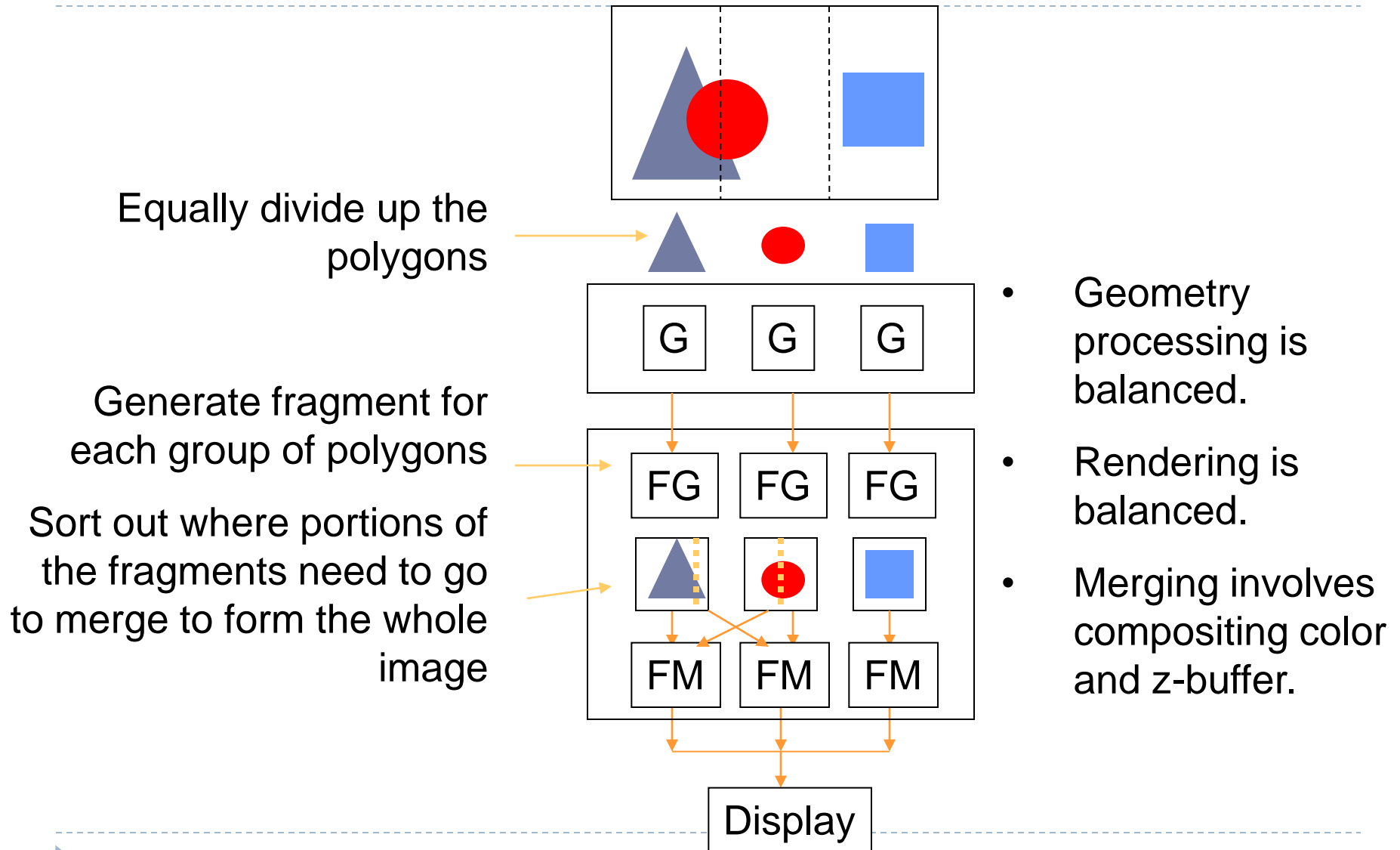
- ▶ Gives the individual triangle pixels a color



Imagine this is my screen and the polygons that will occupy my screen



How Polygons Are Processed (Sort-Last Fragment)



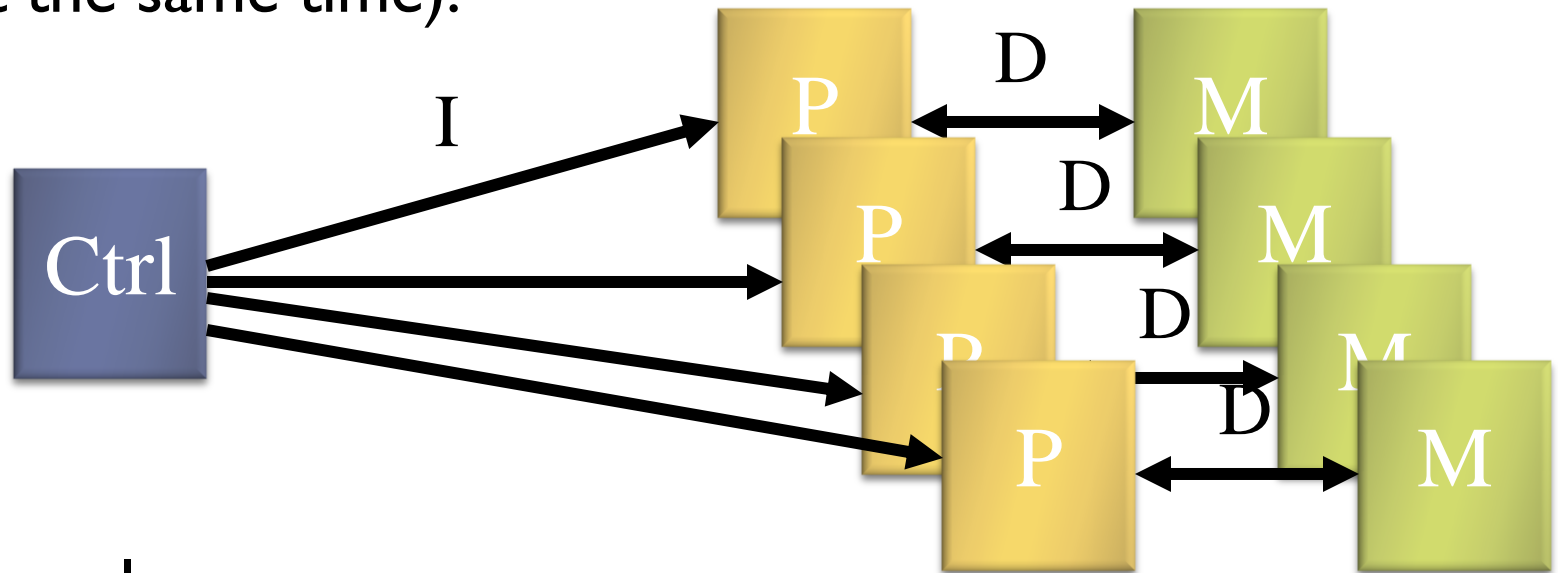
Overview

- ▶ 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.

SIMD (revisited)

- ▶ 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

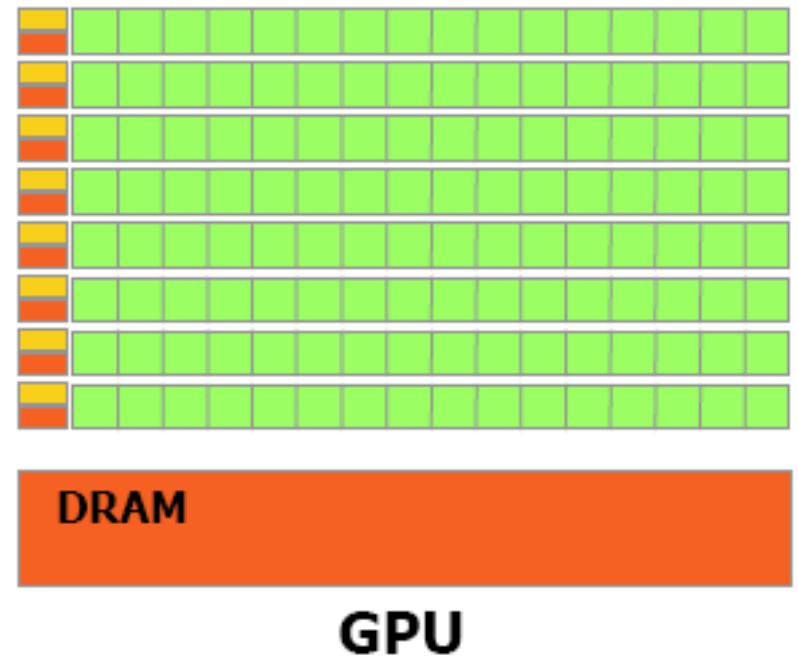
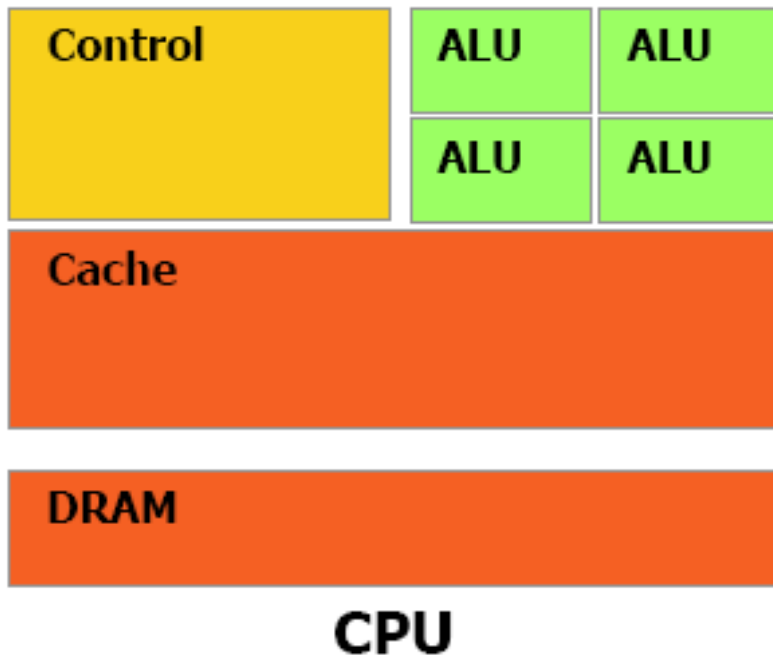


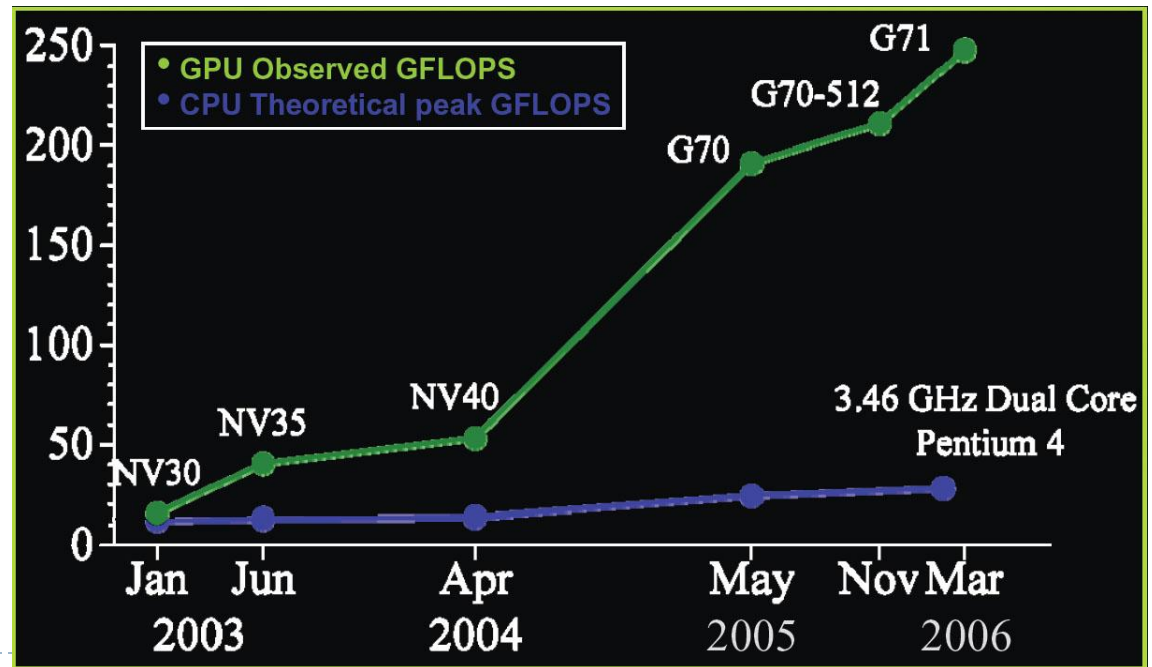
Figure 1-2. The GPU Devotes More Transistors to Data Processing



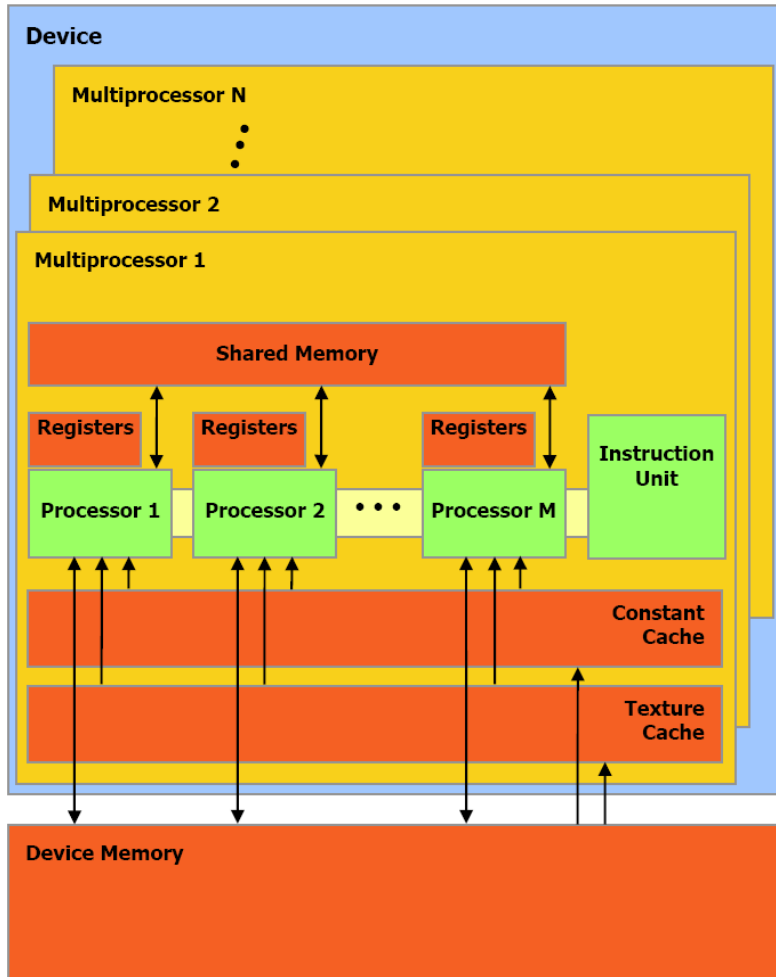
GPU Case: nVidia G80 Architecture

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

Programming Interface

- ▶ Interface to GPU via nVidia's proprietary API – CUDA (very C-like)
- ▶ Looks a lot like UPC (simplified CUDA below)

```
void AddVectors(float *r, float *a, float *a)
{
    int tx = threadIdx.x;           //~processor rank
    r[tx] = a[tx] + b[tx];         //executed in parallel
}
```



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

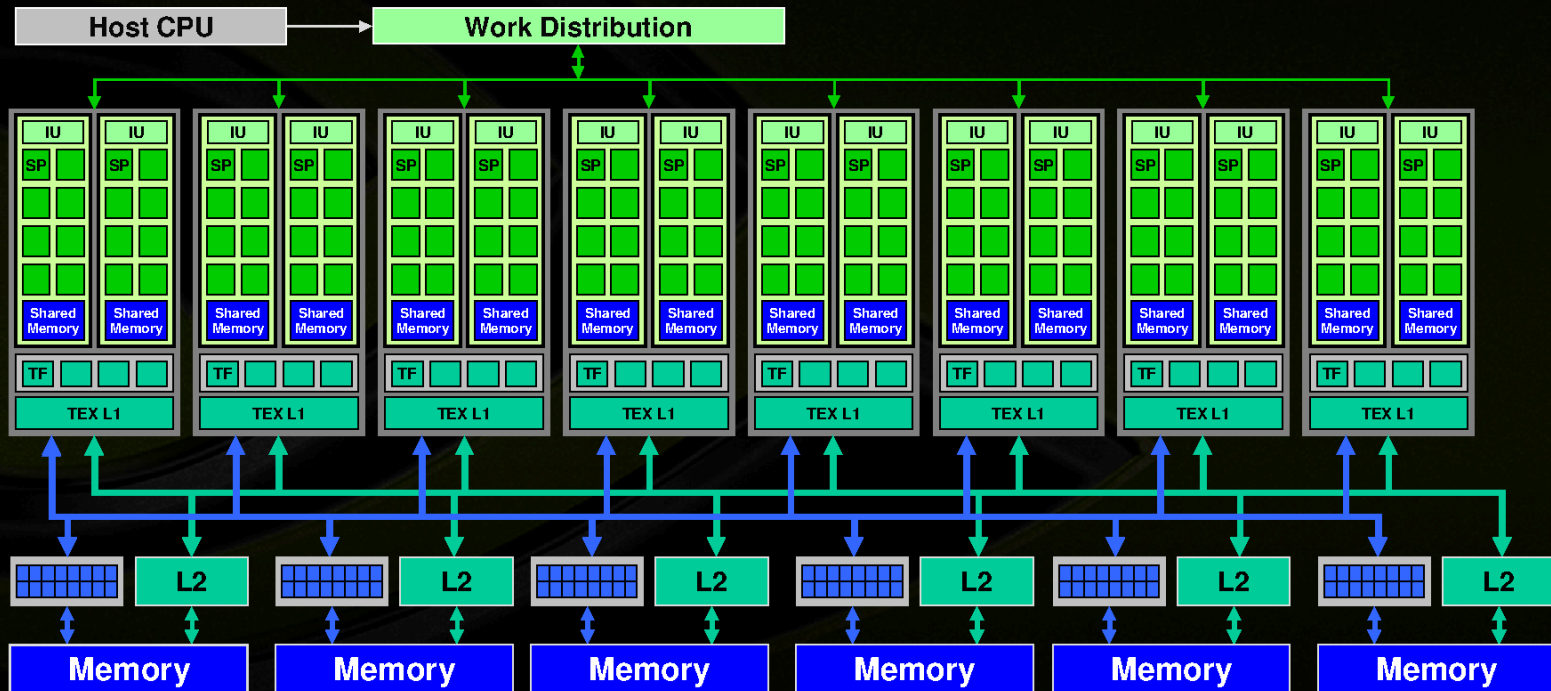
- ▶ Double Precision (need to test performance)
- ▶ Logical Operations on Integer Data
- ▶ Branching-Intensive Operations
- ▶ Random Access, Memory-Intensive Operations



The Future of Computing is Parallel



- CPU clock rate growth is slowing, future speed growth will be from parallelism
- GeForce-8 Series is a massively parallel computing platform
 - 12,288 concurrent threads, hardware managed
 - 128 SP Thread Processor cores at 1.35 GHz == 518 GFLOPS peak
 - GPU Computing features enable C on Graphics Processing Unit



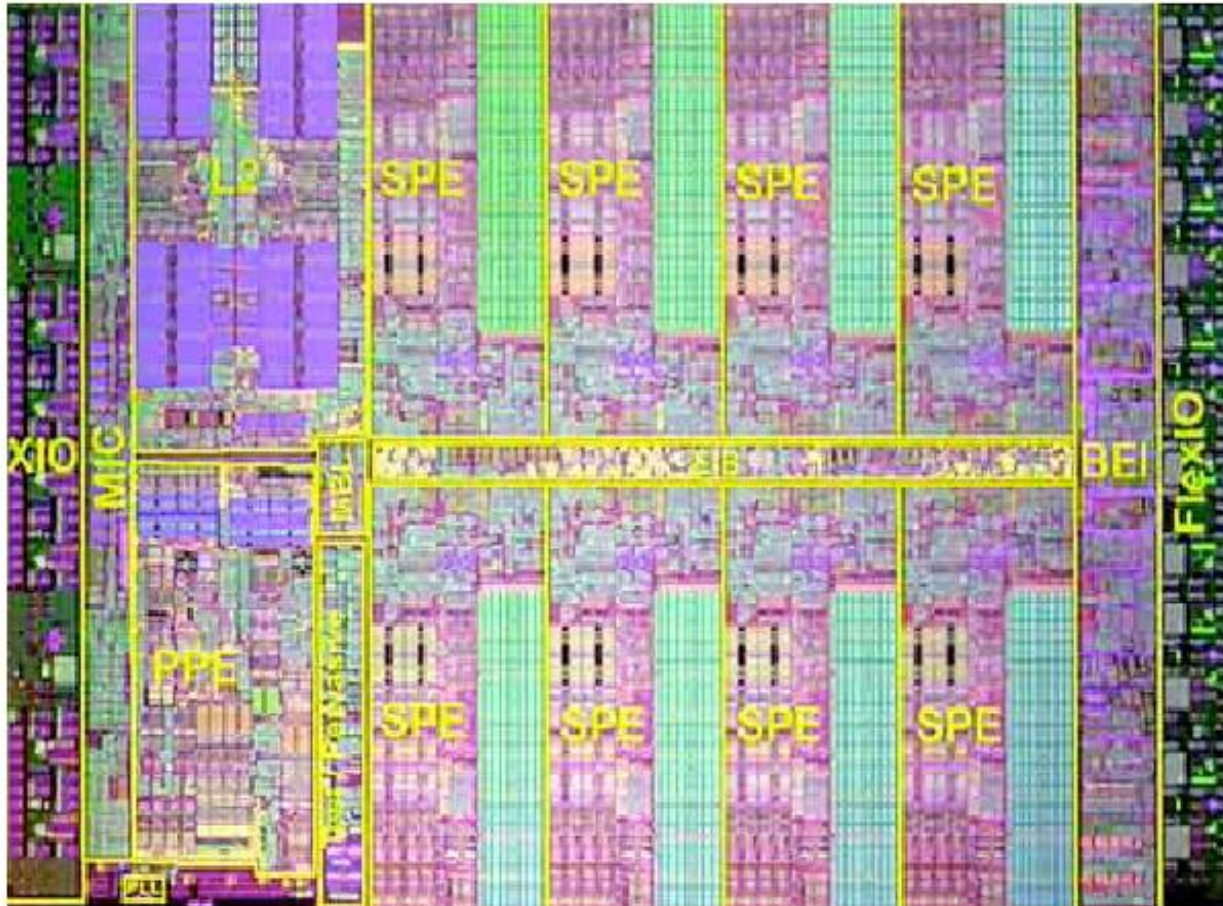
Source: Kirk, "Parallel Computing: What has changed lately?"

GPU Case: Cell Architecture

History

- ▶ Idea generated by SCEI in 1999 after release of PS2
- ▶ STI group (Sony, Toshiba, IBM) formed in 2000
- ▶ In 2001 the first design center opened in the US
- ▶ Fall 2002 US patent released
- ▶ Since then prototypes have been developed and clocked over @4.5 GHz
- ▶ February 2005 final architecture revealed to public
- ▶ In 2005 announced that first commercial product of the Cell will be released in 2006

Cell Architecture Overview

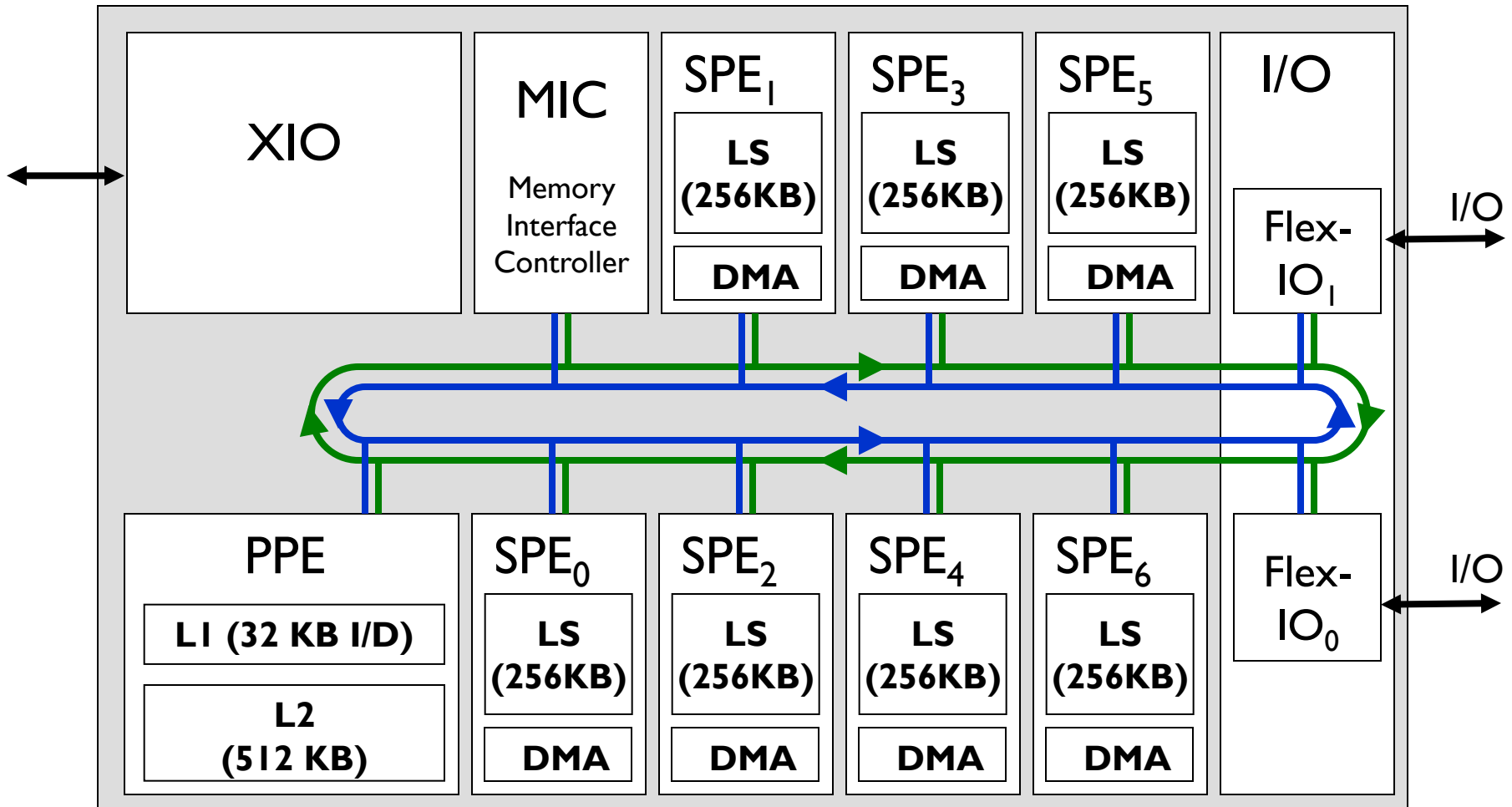


Cell Architecture Overview

- ▶ Intended to be configurable
- ▶ Basic Configuration consists of:
 - ▶ 1 PowerPC Processing Element (PPE)
 - ▶ 8 Synergistic Processing Elements (SPE)
 - ▶ Element Interconnect Bus (EIB)
 - ▶ Rambus Memory Interface Controller (MIC)
 - ▶ Rambus FlexIO interface
 - ▶ 512 KB system Level 2 cache



The Cell Processor



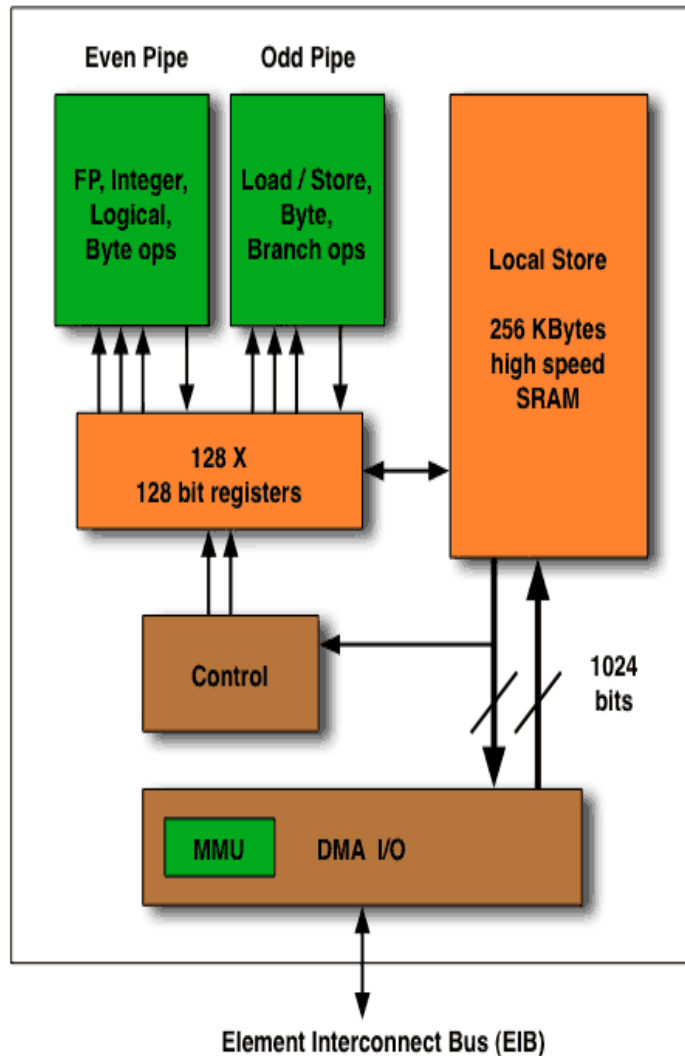
Source: Perthuis, "Introduction to the graphics pipeline of the PS3"

Power Processing Element (PPE)

- ▶ Act as the host processor and performs scheduling for the SPE
- ▶ 64-bit processor based on IBM POWER architecture
(**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC)
- ▶ Dual threaded, in-order execution
- ▶ 32 KB Level 1 cache, connected to 512 KB system level 2 cache
- ▶ Contains VMX (AltiVec) unit and IBM hypervisor technology to allow two operating systems to run concurrently (Such as Linux and a real-time OS for gaming)



Synergistic Processing Unit (SPU)



- ▶ SIMD vector processor and acts independently
- ▶ Handles most of the computational workload
- ▶ Again in-order execution but dual issue*
- ▶ Contains 256 KB local store memory
- ▶ Contains 128 X 128 bit registers

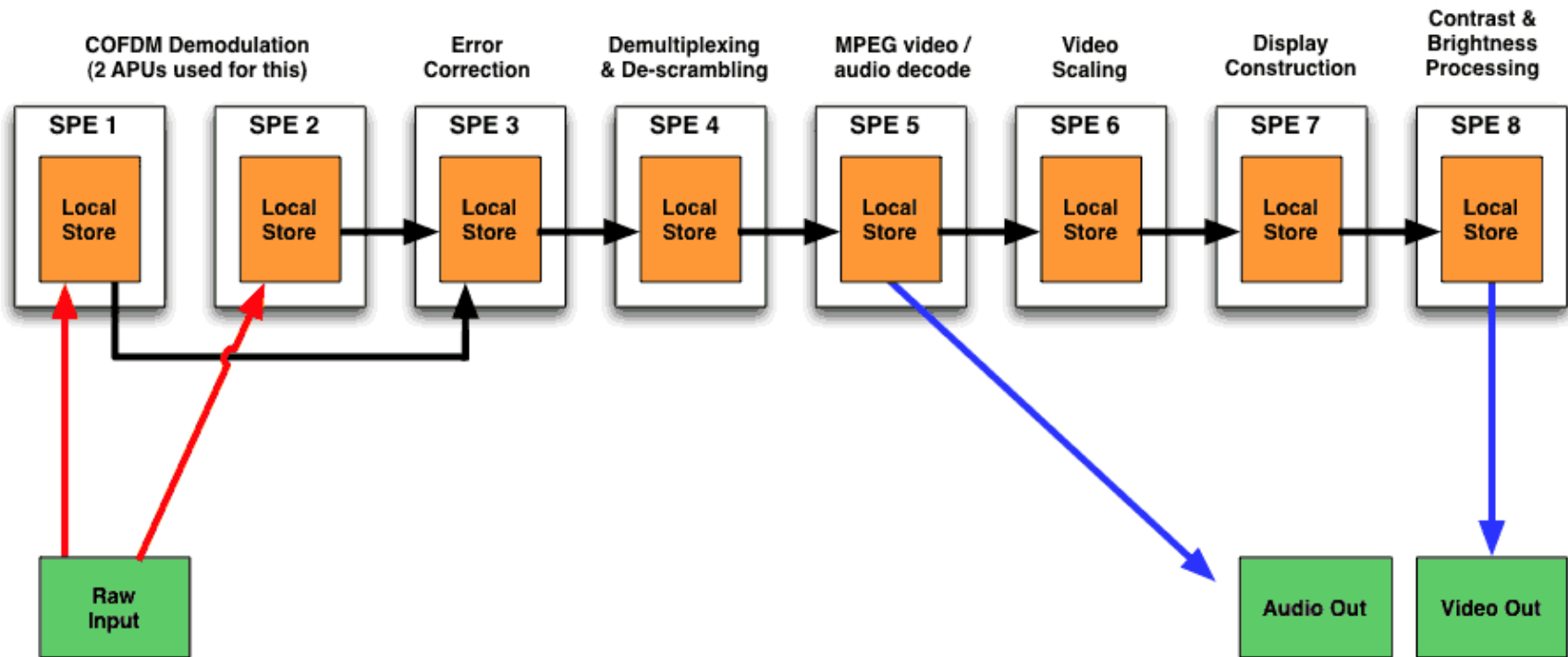
Synergistic Processing Unit (SPU)

- ▶ Operate on registers which are read from or written to local stores.
- ▶ SPE cannot act directly on main memory; they have to move data to and from the local stores.
- ▶ DMA device in SPEs handles moving data between the main memory and the local store.
- ▶ Local Store addresses are aliased in the PPE address map and transfers to and from Local Store to memory (including other Local Stores) are coherent in the system



Stream Processing

Decoding digital TV is a complex process
but it can be broken into a stream



Sony's PS3



PS3 Specs

- ▶ Cell processor @ 3.2 Ghz
- ▶ 7 functional SPE
- ▶ Total 218 SP GFLOPS
- ▶ nVidia RSX GPU (1.8 TFLOPS)
- ▶ 256 MB XDR RAM
- ▶ 256MB GDDR3 VRAM
- ▶ Up to 7 Bluetooth controllers
- ▶ Backwards compatible, WiFi capabilities with PSP



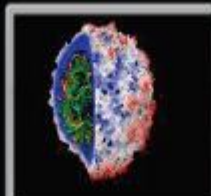
Parallel Programming with CUDA

Applications in several fields



146X

Interactive visualization of volumetric white matter connectivity



36X

Ionic placement for molecular dynamics simulation on GPU



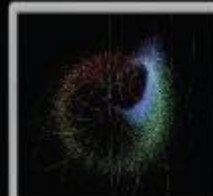
19X

Transcoding HD video stream to H.264



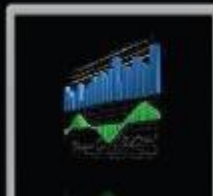
17X

Simulation in Matlab using .mex file CUDA function



100X

Astrophysics N-body simulation



149X

Financial simulation of LIBOR model with swaptions



47X

GLAME@lab: An M-script API for linear Algebra operations on GPU



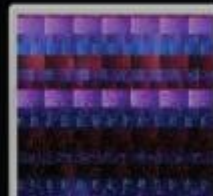
20X

Ultrasound medical imaging for cancer diagnostics



24X

Highly optimized object oriented molecular dynamics



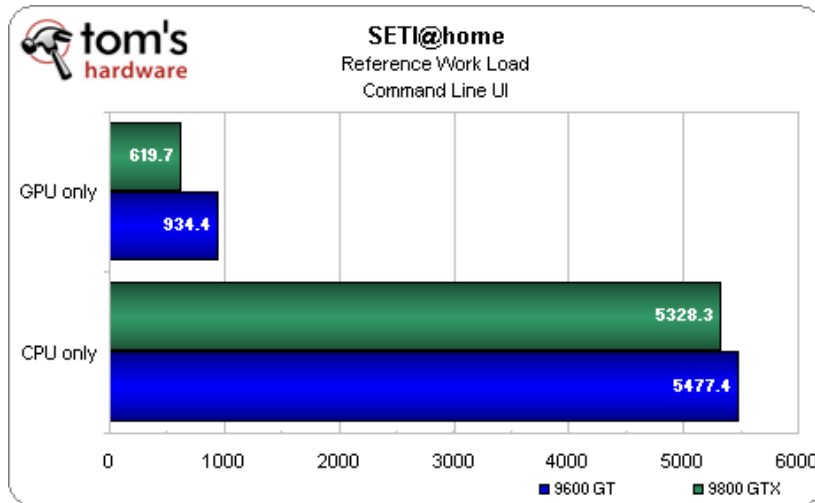
30X

Cmatch exact string matching to find similar proteins and gene sequences

© NVIDIA Corporation 2008

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

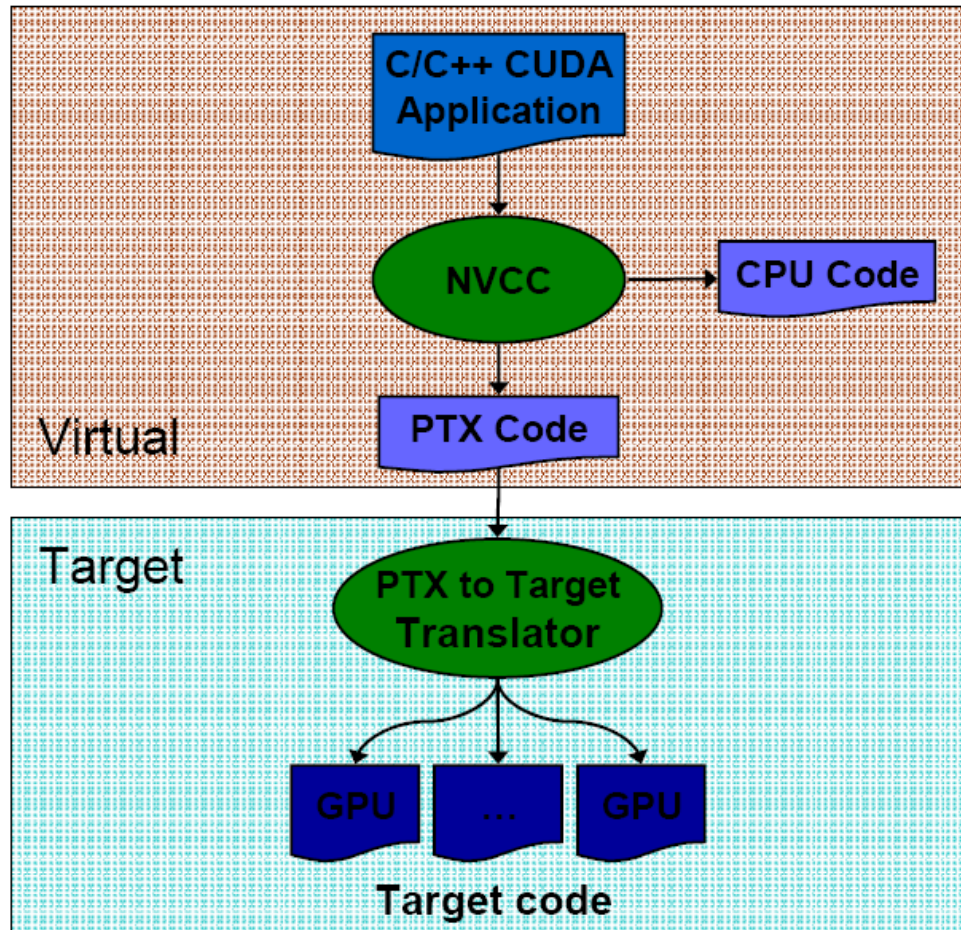


CUDA Concept

- ▶ 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];
}

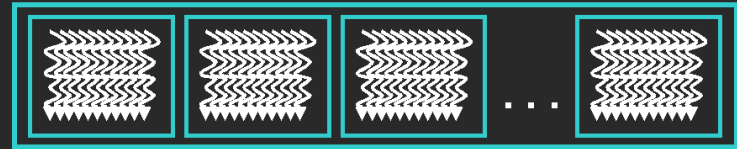
void main()
{
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```



Serial Code

Parallel Kernel

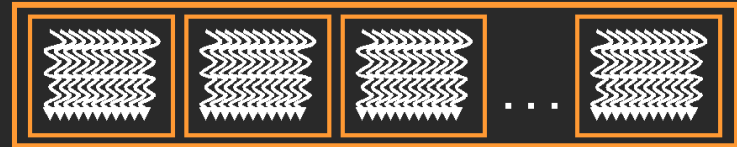
`KernelA<<< nBlk, nTid >>>(args);`



Serial Code

Parallel Kernel

`KernelB<<< nBlk, nTid >>>(args);`



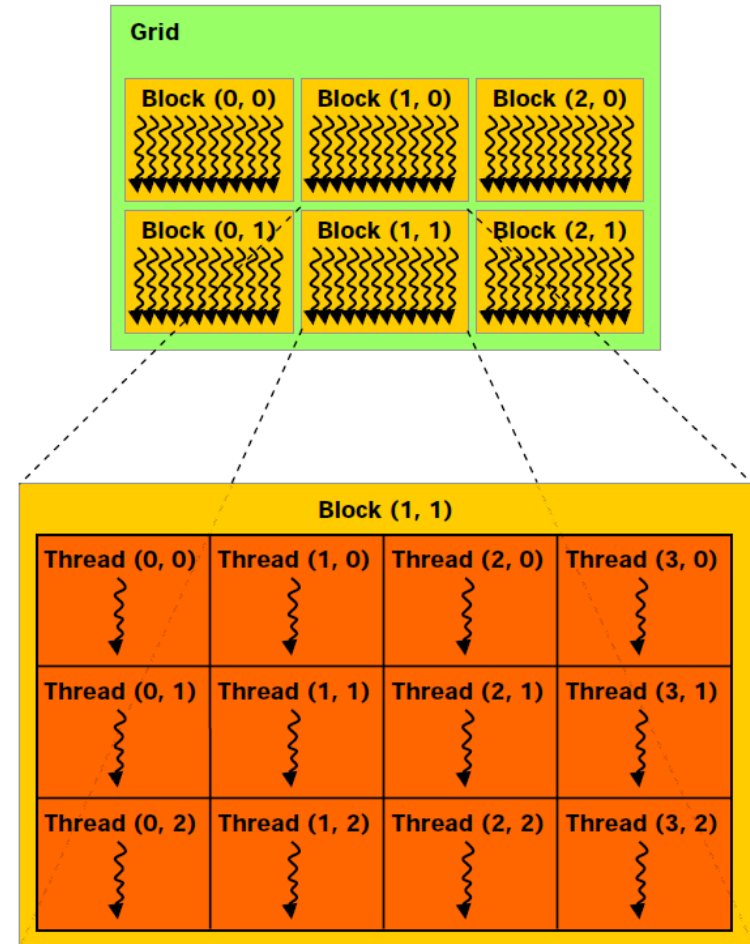
threadID

0	1	2	3	4	5	6	7
---	---	---	---	---	---	---	---

```
...  
float x = input[threadID];  
float y = func(x);  
output[threadID] = y;  
...
```

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



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
```

```
}
```

```
__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
```

```
}
```

```
__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
```

```
}
```



Incremental Array Example

CPU Program

```
void inc_cpu(int *a, int N)
{
    int idx;

    for (idx = 0; idx < N; idx++)
        a[idx] = a[idx] + 1;
}

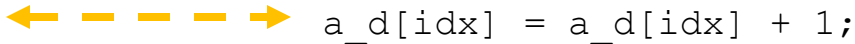
void main()
{
    ...
    inc_cpu(a, N);
    ...
}
```

CUDA Program

```
__global__ void inc_gpu(int *a_d, int N)
{
    int idx = blockIdx.x * blockDim.x
              + threadIdx.x;

    if (idx < N)
        a_d[idx] = a_d[idx] + 1;
}

void main()
{
    ...
    dim3 dimBlock (blocksize);
    dim3 dimGrid(ceil(N/(float)blocksize));
    inc_gpu<<<dimGrid, dimBlock>>>(a_d, N);
    ...
}
```



Incremental Array Example

- ▶ Increment N-element vector a by scalar b



- ▶ Let's assume $N=16$, $\text{blockDim}=4 \rightarrow 4$ blocks



$\text{blockIdx.x}=0$
 $\text{blockDim.x}=4$
 $\text{threadIdx.x}=0,1,2,3$
 $\text{idx}=0,1,2,3$



$\text{blockIdx.x}=1$
 $\text{blockDim.x}=4$
 $\text{threadIdx.x}=0,1,2,3$
 $\text{idx}=4,5,6,7$



$\text{blockIdx.x}=2$
 $\text{blockDim.x}=4$
 $\text{threadIdx.x}=0,1,2,3$
 $\text{idx}=8,9,10,11$



$\text{blockIdx.x}=3$
 $\text{blockDim.x}=4$
 $\text{threadIdx.x}=0,1,2,3$
 $\text{idx}=12,13,14,15$

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

will map from local index `threadIdx` to global index

NB: `blockDim` should be bigger than 4 in real code, this is just an example



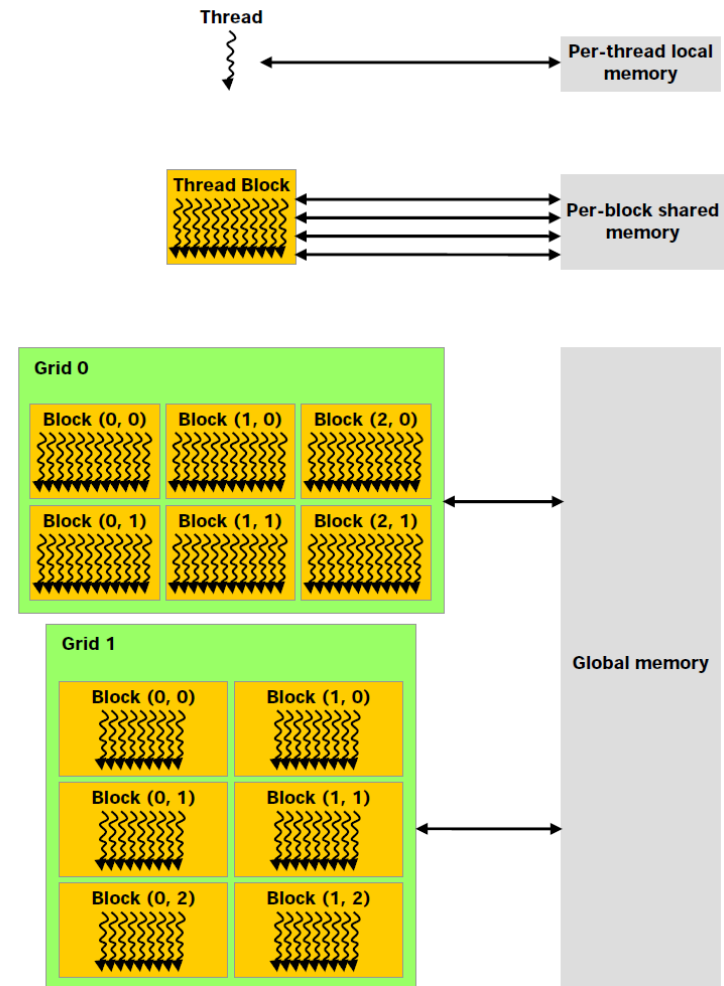
Note on CUDA Kernel

- ▶ **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**



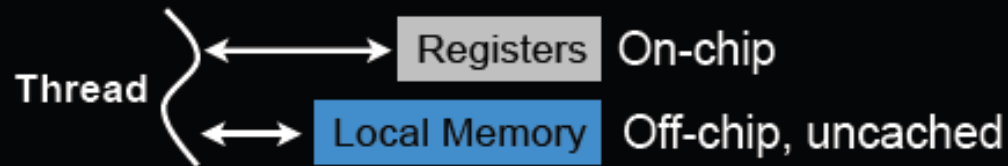
CUDA Memory Hierarchy

- ▶ 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

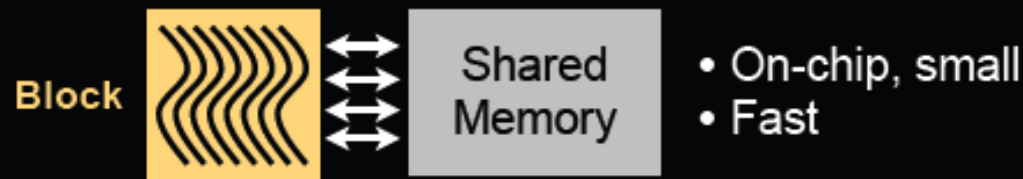


Kernel Memory Access

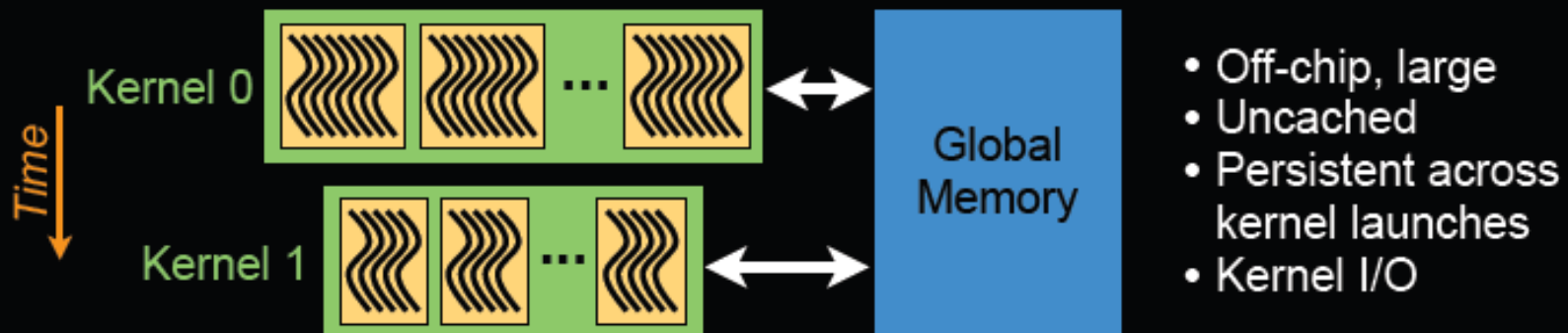
● Per-thread



● Per-block

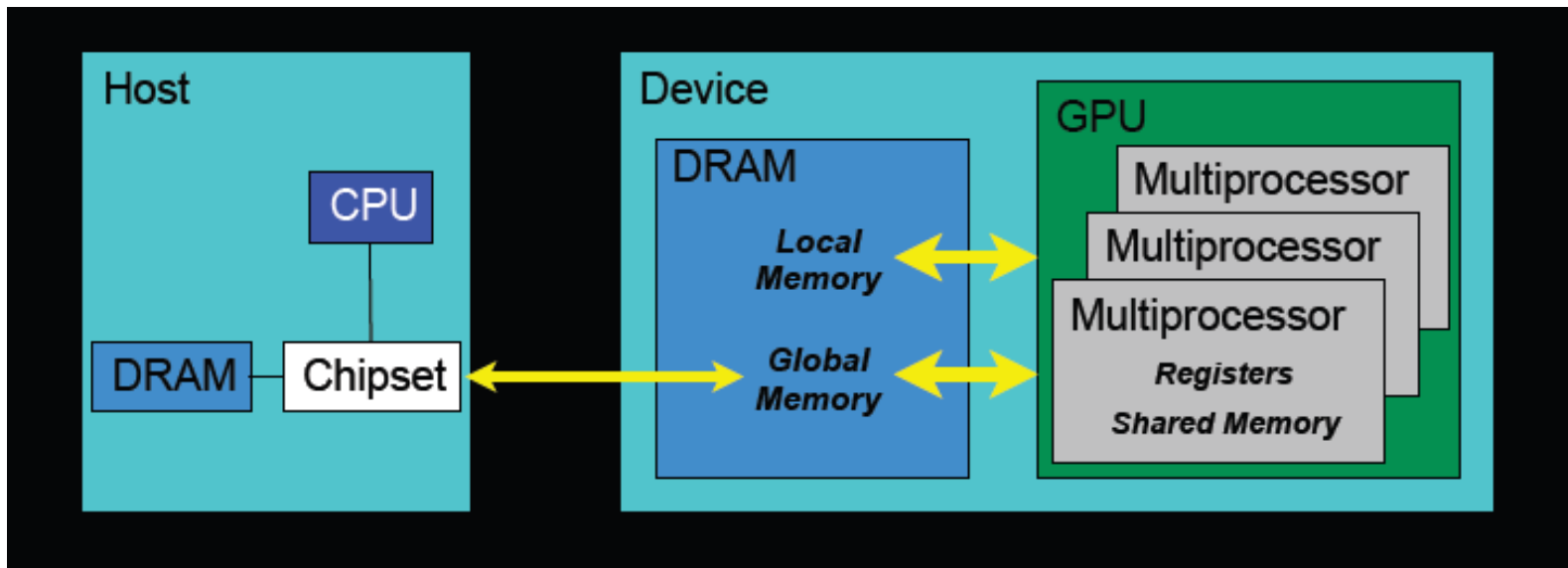


● Per-device



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



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

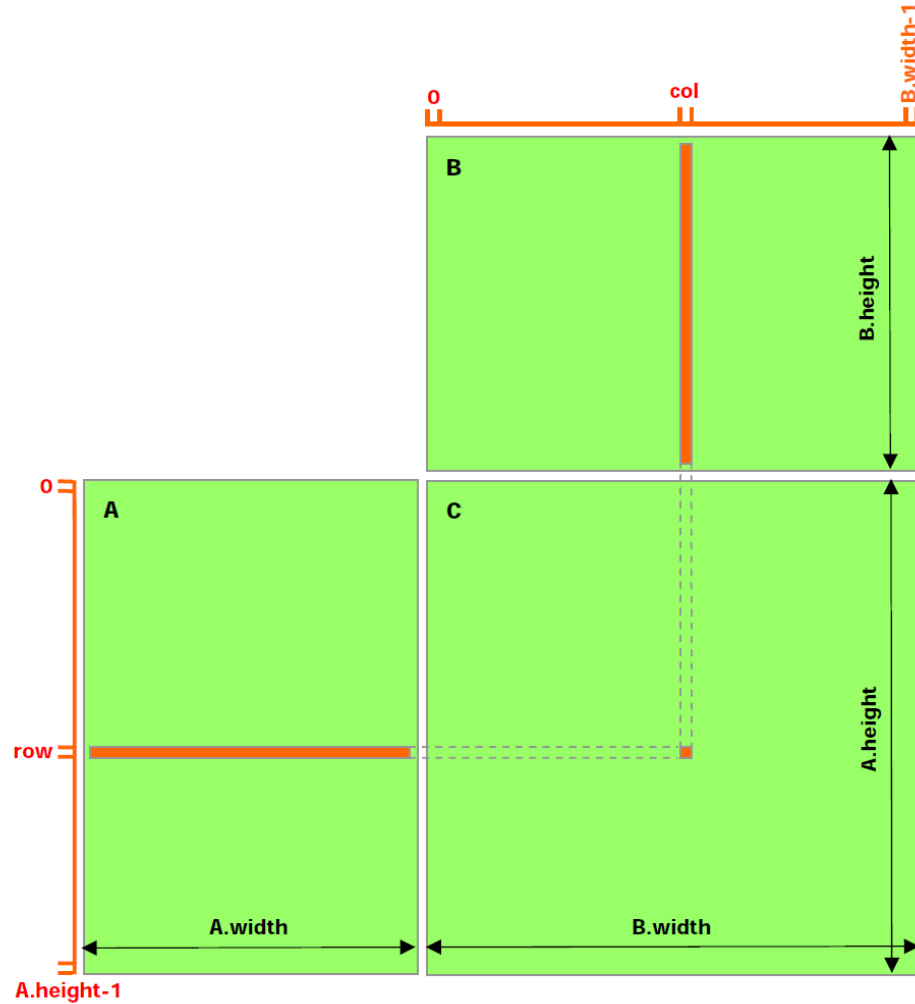
```
__global__ void kernel(...)  
{  
    ...  
    __shared__ float sData[256];  
    ...  
}  
int main(void)  
{  
    ...  
    kernel<<<nBlocks, blockSize>>> (...);  
    ...  
}
```

Size known at kernel launch

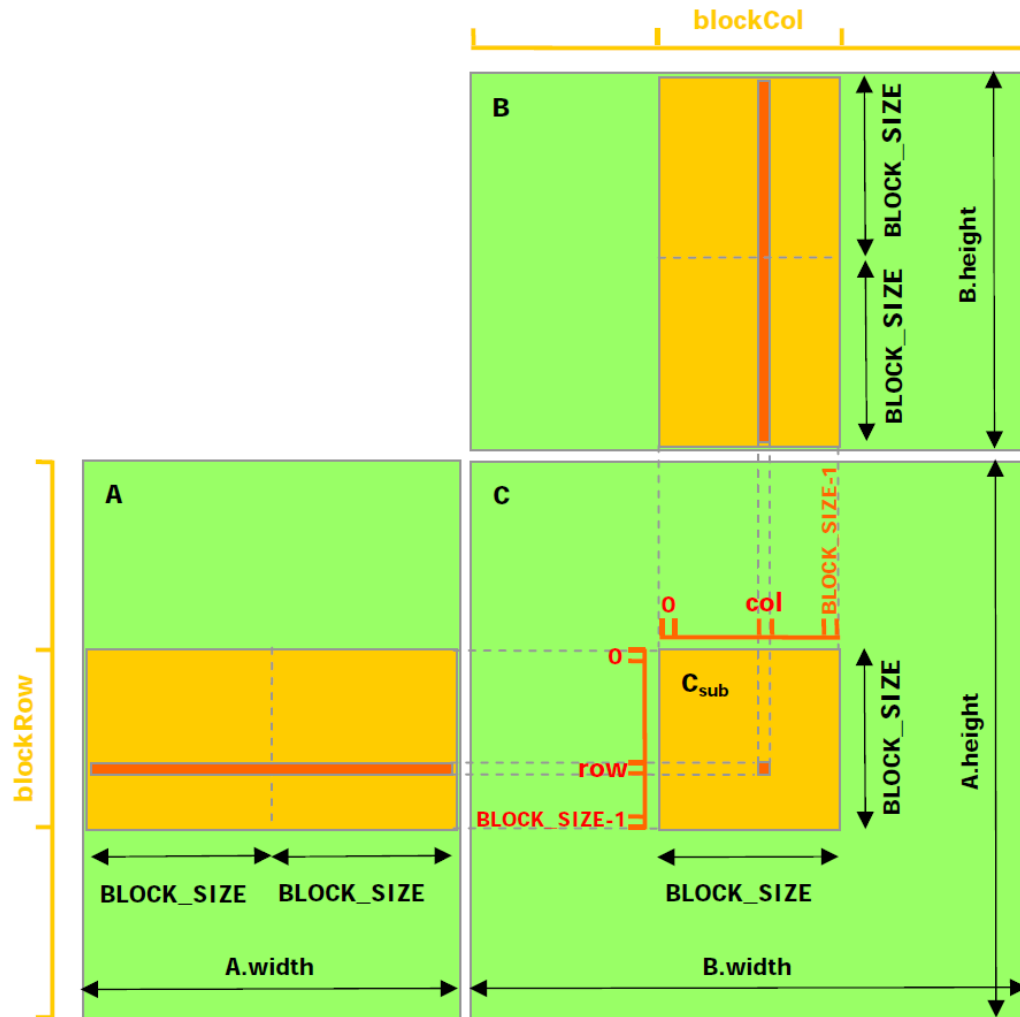
```
__global__ void kernel(...)  
{  
    ...  
    extern __shared__ float sData[];  
    ...  
}  
int main(void)  
{  
    ...  
    smBytes=blockSize*sizeof(float);  
    kernel<<<nBlocks, blockSize,  
        smBytes>>> (...);  
    ...  
}
```



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
- ▶ Due date: 19 February 2010 at 18:00
- ▶ How to submit: sending email to “natawut.n@chula.ac.th”
- ▶ Note: I will use timestamp on your email

