The CUDA Programming Model
Overview

- Massively Parallel Processing
- CPU/GPU Architecture
- GPUs
- CUDA
- OpenCL
- APU
- Accelerators - MIC
Why Massively Parallel Processing?

- A quiet revolution and potential build-up
  - Calculation: TFLOPS vs. 100 GFLOPS
  - Memory Bandwidth: ~10x

- GPU in every PC—massive volume and potential impact
# NVIDIA Tesla Family

<table>
<thead>
<tr>
<th>NVIDIA Tesla Family Specification Comparison</th>
<th>Tesla K80</th>
<th>Tesla K40</th>
<th>Tesla K20X</th>
<th>Tesla K20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stream Processors</td>
<td>2 x 2496</td>
<td>2880</td>
<td>2688</td>
<td>2496</td>
</tr>
<tr>
<td>Core Clock</td>
<td>562MHz</td>
<td>745MHz</td>
<td>732MHz</td>
<td>706MHz</td>
</tr>
<tr>
<td>Boost Clock(s)</td>
<td>875MHz</td>
<td>810MHz, 875MHz</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Memory Clock</td>
<td>5GHz GDDR5</td>
<td>6GHz GDDR5</td>
<td>5.2GHz GDDR5</td>
<td>5.2GHz GDDR5</td>
</tr>
<tr>
<td>Memory Bus Width</td>
<td>2 x 384-bit</td>
<td>384-bit</td>
<td>384-bit</td>
<td>320-bit</td>
</tr>
<tr>
<td>VRAM</td>
<td>2 x 12GB</td>
<td>12GB</td>
<td>6GB</td>
<td>5GB</td>
</tr>
<tr>
<td>Single Precision</td>
<td>8.74 TFLOPS</td>
<td>4.29 TFLOPS</td>
<td>3.95 TFLOPS</td>
<td>3.52 TFLOPS</td>
</tr>
<tr>
<td>Double Precision</td>
<td>2.91 TFLOPS (1/3)</td>
<td>1.43 TFLOPS (1/3)</td>
<td>1.31 TFLOPS (1/3)</td>
<td>1.17 TFLOPS (1/3)</td>
</tr>
<tr>
<td>Transistor Count</td>
<td>2 x 7.1B(?)</td>
<td>7.1B</td>
<td>7.1B</td>
<td>7.1B</td>
</tr>
<tr>
<td>TDP</td>
<td>300W</td>
<td>235W</td>
<td>235W</td>
<td>225W</td>
</tr>
<tr>
<td>Cooling</td>
<td>Passive</td>
<td>Active/Passive</td>
<td>Passive</td>
<td>Active/Passive</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>TSMC 28nm</td>
<td>TSMC 28nm</td>
<td>TSMC 28nm</td>
<td>TSMC 28nm</td>
</tr>
<tr>
<td>Architecture</td>
<td>Kepler</td>
<td>Kepler</td>
<td>Kepler</td>
<td>Kepler</td>
</tr>
<tr>
<td>Launch Price</td>
<td>$5000</td>
<td>$5499</td>
<td>~$3799</td>
<td>~$3299</td>
</tr>
</tbody>
</table>
NVIDIA Tesla K80

NVIDIA TESLA ACCELERATOR PERFORMANCE

Throughput

X-times

GROMACS  LAMMPS  NAMD  AMBER14  HOOMD-BLUE  CP2K  QUANTUM ESPRESSO  LSMS  MINIFE (CGTIME)  SPECFEM3D  CLOVERLEAF  MILC  CHROMA  RTM  LINPACK  CAFFE

COMPUTATIONAL CHEMISTRY AND MOLECULAR DYNAMICS  MATERIALS SCIENCE  PHYSICS  SEISMIC PROCESSING  BENCHMARK  MACHINE LEARNING
GeForce 8800 (2007)

16 highly threaded SM’s,
>128 FPU’s,
367 GFLOPS,
768 MB DRAM,
86.4 GB/S Mem BW,
4GB/S BW to CPU
Fermi (2010)

~1.5TFLOPS (SP)/~800GFLOPS (DP)
230 GB/s DRAM Bandwidth
Future Apps Reflect a Concurrent World

- Exciting applications in future mass computing market have been traditionally considered “supercomputing applications”
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These “Super-apps” represent and model physical, concurrent world
- Various granularities of parallelism exist, but…
  - programming model must not hinder parallel implementation
  - data delivery needs careful management
Stretching Traditional Architectures

- Traditional parallel architectures cover some super-applications
  - DSP, GPU, network apps, Scientific

- The game is to grow mainstream architectures “out” or domain-specific architectures “in”
  - CUDA is latter
Speedup of Applications

- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
Classic PC architecture

- Northbridge connects 3 components that must communicate at high speed:
  - CPU, DRAM, video
  - Video also needs to have 1st-class access to DRAM
  - Previous NVIDIA cards are connected to AGP, up to 2 GB/s transfers

- Southbridge serves as a concentrator for slower I/O devices
(Original) PCI Bus Specification

- Connected to the southBridge
  - Originally 33 MHz, 32-bit wide, 132 MB/second peak transfer rate
  - More recently 66 MHz, 64-bit, 512 MB/second peak
  - Upstream bandwidth remain slow for device (256MB/s peak)
  - Shared bus with arbitration
    - Winner of arbitration becomes bus master and can connect to CPU or DRAM through the southbridge and northbridge
PCI as Memory Mapped I/O

- PCI device registers are mapped into the CPU’s physical address space
  - Accessed through loads/stores (kernel mode)
- Addresses assigned to the PCI devices at boot time
  - All devices listen for their addresses
PCI Express (PCIe)

- Switched, point-to-point connection
  - Each card has a dedicated “link” to the central switch, no bus arbitration.
  - Packet switches messages form virtual channel
  - Prioritized packets for QoS
    - E.g., real-time video streaming
PCIe PC Architecture

- PCIe forms the interconnect backbone
  - Northbridge/Southbridge are both PCIe switches
  - Some Southbridge designs have built-in PCI-PCIe bridge to allow old PCI cards
  - Some PCIe cards are PCI cards with a PCI-PCIe bridge

- Source: Jon Stokes, PCI Express: An Overview
  - http://arstechnica.com/articles/paediatric/hardware/pcie.ars
Intel Skylake Processor (2015)
Intel Skylake Processor (2015)
CUDA Overview

- CUDA (Compute Unified Device Architecture)
- Programming model – basic concepts and data types
- CUDA application programming interface - basic
- Simple examples to illustrate basic concepts and functionalities
- Performance features will be covered later
CUDA – C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host C code**
  - Highly parallel parts in **device SPMD kernel C code**

```
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```
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
G80 CUDA mode – A Device Example

- Processors execute computing threads
- New operating mode/HW interface for computing

Diagram:

- Host
  - Input Assembler
  - Thread Execution Manager

- Parallel Data Cache
- Texture

- Load/store

Global Memory
## Extended C

- **Type Qualifiers**
  - global, device, shared, local, constant

- **Keywords**
  - threadIdx, blockIdx

- **Intrinsics**
  - __syncthreads

- **Runtime API**
  - Memory, symbol, execution management

- **Function launch**

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
    __shared__ float region[M];
    ...

    region[threadIdx] = image[i];

    __syncthreads()
    ...

    image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<100, 10>>> (myimage);
```
**NVCC Compiler’s Role: Code/Compile Device**

`mycode.cu`

```c
int main_data;
__shared__ int sdata;

Main() {
    __host__ hfunc () {
        int hdata;
        <<<gfnc(g,b,m)>>>();
    }

    __global__ gfunc() {
        int gdata;
    }

    __device__ dfunc() {
        int ddata;
    }
```

Compiled by native compiler: gcc, icc, cc

```c
int main_data;
__shared__ sdata;

Main() {
    __host__ hfunc () {
        int hdata;
        <<<gfnc(g,b,m)>>>();
    }

    __global__ gfunc() {
        int gdata;
    }

    __device__ dfunc() {
        int ddata;
    }
```

Compiled by nvcc compiler
Mark Murphy, “NVIDIA’s Experience with Open64,”
www.capsl.udel.edu/conferences/open64/2008/Papers/101.doc
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
- All threads run the same code (SPMD)
- Each thread has an ID that it uses to compute memory addresses and make control decisions

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

threadID 0 1 2 3 4 5 6 7

...
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
Block IDs and Thread IDs

- Each thread uses IDs to decide what data to work on
  - Block ID: 1D or 2D
  - Thread ID: 1D, 2D, or 3D

- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …
CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come later
CUDA API Highlights: Easy and Lightweight

- The API is an extension to the ANSI C programming language
  - Low learning curve

- The hardware is designed to enable lightweight runtime and driver
  - High performance
CUDA Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - Address of a pointer to the allocated object
    - Size of allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
CUDA Device Memory Allocation (cont.)

- **Code example:**
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md
  - “d” is often used to indicate a device data structure

```
TILE_WIDTH = 64;
Float* Md;
int size = TILE_WIDTH * TILE_WIDTH * sizeof(float);

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```
CUDA Host-Device Data Transfer

- 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

- Asynchronous transfer
CUDA Host-Device Data Transfer

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

```c
cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);

cudaMemcpy(M, Md, size, cudaMemcpyDeviceToHost);
```
CUDA Function Declarations

<table>
<thead>
<tr>
<th><strong>device</strong></th>
<th><strong>global</strong></th>
<th><strong>host</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>float DeviceFunc()</td>
<td>void KernelFunc()</td>
<td>float HostFunc()</td>
</tr>
</tbody>
</table>

- **__global__** defines a kernel function
  - Must return **void**
- **__device__** and **__host__** can be used together
CUDA Function Declarations (cont.)

- __device__ functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
A kernel function must be called with an execution configuration:

```c
__global__ void KernelFunc(...);
dim3 DimGrid(100, 50); // 5000 thread blocks
dim3 DimBlock(4, 8, 8); // 256 threads per block
size_t SharedMemBytes = 64; // 64 bytes of shared memory
KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
```
Simple working code example

- **Goal for this example:**
  - Really simple but illustrative of key concepts
  - Fits in one file with simple compile command
  - Can absorb during lecture

- **What does it do?**
  - Scan elements of array of numbers (any of 0 to 9)
  - How many times does “6” appear?
  - Array of 16 elements, each thread examines 4 elements, 1 block in grid, 1 grid

threadIdx.x = 0 examines in_array elements 0, 4, 8, 12
threadIdx.x = 1 examines in_array elements 1, 5, 9, 13
threadIdx.x = 2 examines in_array elements 2, 6, 10, 14
threadIdx.x = 3 examines in_array elements 3, 7, 11, 15

Known as a cyclic data distribution
CUDA Pseudo-Code

**MAIN PROGRAM:**

Initialization
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

**GLOBAL FUNCTION:**

Thread scans subset of array elements

Call *device* function to compare with “6”

Compute local result

**HOST FUNCTION:**

Allocate memory on device for copy of input and output

Copy input to *device*

Set up grid/block

Call *global* function

Synchronize after completion

Copy *device* output to host

**DEVICE FUNCTION:**

Compare current element and “6”

Return 1 if same, else 0
Main Program: Preliminaries

MAIN PROGRAM:

Initialization
• Allocate memory on host for input and output
• Assign random numbers to input array

Call host function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    ...
}
```
**MAIN PROGRAM:**

Initialization *(OMIT)*
- Allocate memory on host for input and output
- Assign random numbers to input array

Call **host** function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
  int *in_array, *out_array;
  /* initialization */ …
  outer_compute(in_array, out_array);
  …
}
```
**Main Program: Calculate Output & Print Result**

**MAIN PROGRAM:**

Initialization (OMIT)
- Allocate memory on host for input and output
- Assign random numbers to input array

Call *host* function

Calculate final output from per-thread output

Print result

```c
#include <stdio.h>
#define SIZE 16
#define BLOCKSIZE 4
__host__ void outer_compute (int *in_arr, int *out_arr);

int main(int argc, char **argv)
{
    int *in_array, *out_array;
    int sum = 0;
    /* initialization */ …
    outer_compute(in_array, out_array);
    for (int i=0; i<BLOCKSIZE; i++) {
        sum+=out_array[i];
    }
    printf ("Result = %d\n",sum);
}
```
HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

__host__ void outer_compute (int *h_in_array, int *h_out_array)
{
    int *d_in_array, *d_out_array;
    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    ...
HOST FUNCTION:
Allocate memory on device for copy of input and output
Copy input to device
Set up grid/block
Call global function
Synchronize after completion
Copy device output to host

_host__ void outer_compute (int *h_in_array, int *h_out_array) {
int *d_in_array, *d_out_array;

cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
... do computation ...
cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
HOST FUNCTION:

Allocate memory on device for copy of *input* and *output*

Copy input to *device*

Set up grid/block

Call *global* function

Synchronize after completion

Copy *device* output to host

```c
__host__ void outer_compute (int *h_in_array, int *h_out_array) {
    int *d_in_array, *d_out_array;

    cudaMalloc((void **) &d_in_array, SIZE*sizeof(int));
    cudaMalloc((void **) &d_out_array, BLOCKSIZE*sizeof(int));
    cudaMemcpy(d_in_array, h_in_array, SIZE*sizeof(int), cudaMemcpyHostToDevice);
    compute<<<(1,BLOCKSIZE)>>>(d_in_array, d_out_array);
    cudaMemcpy(h_out_array, d_out_array, BLOCKSIZE*sizeof(int), cudaMemcpyDeviceToHost);
}
```
GLOBAL FUNCTION:

Thread scans subset of array elements

Call *device* function to compare with “6”

Compute local result

```
__global__ void compute(int *d_in, int *d_out)
{
    d_out[threadIdx.x] = 0;
    for (int i=0; i<SIZE/BLOCKSIZE; i++)
    {
        int val = d_in[i*BLOCKSIZE + threadIdx.x];
        d_out[threadIdx.x] += compare(val, 6);
    }
}
```
Device Function

DEVICE FUNCTION:

Compare current element and “6”
Return 1 if same, else 0

```c
__device__ int compare(int a, int b)
{
    if (a == b) return 1;
    return 0;
}
```
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

- \( P = M \times N \) of size \( WIDTH \times WIDTH \)
- 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

\[
\begin{array}{cccc}
M_{0,0} & M_{0,1} & M_{0,2} & M_{0,3} \\
M_{1,0} & M_{1,1} & M_{1,2} & M_{1,3} \\
M_{2,0} & M_{2,1} & M_{2,2} & M_{2,3} \\
M_{3,0} & M_{3,1} & M_{3,2} & M_{3,3}
\end{array}
\]
Step 1: Matrix Multiplication A Simple Host Version

// Matrix multiplication on the (CPU) 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)

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

    // 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);
}
```

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

   cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);
   cudaMemcpy(Nd, N, size, cudaMemcpyHostToDevice);

   // Allocate P on the device
   cudaMemcpy(Pd, P, size, cudaMemcpyHostToDevice);
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.)

```cpp
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
Compiling a CUDA Program

C/C++ CUDA Application

float4 me = gx[gtid];
me.x += me.y * me.z;

NVCC

CPU Code

PTX Code

Virtual

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

Physical

PTX to Target Compiler

ld.global.v4.f32
mad.f32

G80

... GPU

Target code

Bilkent University
Linking

- Any executable with CUDA code requires two dynamic libraries:
  - The CUDA runtime library (`cudart`)
  - The CUDA core library (`cuda`)
Debugging Using the Device Emulation Mode

- An executable compiled in device emulation mode (\texttt{nvcc -deviceemu}) runs completely on the host using the CUDA runtime
  - No need of any device and CUDA driver
  - Each device thread is emulated with a host thread

- Running in device emulation mode, one can:
  - Use host native debug support (breakpoints, inspection, etc.)
  - Access any device-specific data from host code and vice-versa
  - Call any host function from device code (e.g. \texttt{printf}) and vice-versa
  - Detect deadlock situations caused by improper usage of \texttt{__syncthreads}
Block Usage in Matrix Multiplication

Block (0,0)  Block (1,0)

Block (0,1)  Block (1,1)

TILE_WIDTH = 2
Block Usage in Matrix Multiplication
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
    // Calculate the row index of the Pd element and M
    int Row = blockIdx.y*TILE_WIDTH + threadIdx.y;
    // Calculate the column index of Pd and N
    int Col = blockIdx.x*TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
        Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];

    Pd[Row*Width+Col] = Pvalue;
}
CUDA Thread Block

- All threads in a block execute the same kernel program (SPMD)
- Programmer declares block:
  - Block size 1 to 512 concurrent threads
  - Block shape 1D, 2D, or 3D
  - Block dimensions in threads
- Threads have thread id numbers within block
  - Thread program uses thread id to select work and address shared data
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks!

Courtesy: John Nickolls, NVIDIA
**Transparent Scalability**

- Hardware is free to assign 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.
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 execution
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
Source Code

- Matrix Multiplication

__syncthreads()

__global__ void globFunction(int *arr, int N) {

    __shared__ int local_array[THREADS_PER_BLOCK];

    //local block memory cache
    int idx = blockIdx.x* blockDim.x+ threadIdx.x;

    //...calculate results
    local_array[threadIdx.x] = results;

    //synchronize the local threads writing to the local memory
    cache __syncthreads();

    // read the results of another thread in the current thread
    int val = local_array[(threadIdx.x + 1) % THREADS_PER_BLOCK];

    //write back the value to global memory
    arr[idx] = val;
}
cudaThreadSynchronize()

- **cudaThreadSynchronize()** is a **_host_** function
  - Waits for all previous async operations (i.e. kernel calls, async memory copies) to complete.

- **__synchtreads()** is a **_device_** function
  - Acts as a thread barrier.
  - All threads in a block must reach the barrier before any can continue execution.
  - It is only of use when you need to avoid race conditions when threads in a block access shared memory.
Use CUDA best practices guide!

CUDA!
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