Graphic Processing Units – GPU (Section 7.7)

History of GPUs

• VGA in early 90’s -- A memory controller and display generator connected to some (video) RAM
• By 1997, VGA controllers were incorporating some acceleration functions
• In 2000, a single chip graphics processor incorporated almost every detail of the traditional high-end workstation graphics pipeline
  - Processors oriented to 3D graphics tasks
  - Vertex/pixel processing, shading, texture mapping, rasterization
• More recently, processor instructions and memory hardware were added to support general-purpose programming languages
• OpenGL: A standard specification defining an API for writing applications that produce 2D and 3D computer graphics
• CUDA (compute unified device architecture): A scalable parallel programming model and language for GPUs based on C/C++
Historical PC architecture
Contemporary PC architecture
Basic unified GPU architecture

SM = streaming multiprocessor
MT Issue
C-Cache
I-Cache
SP
C-Cache
SFU
Shared Memory
SFU = special function unit

ROP = raster operations pipeline
Note: The following slides are extracted from different presentations by NVIDIA (publicly available on the web)

For more details on CUDA see:
http://docs.nvidia.com/cuda/cuda-c-programming-guide

(or search for "CUDA programming guide" on Google)
The World Leader in Parallel Processing
Enter the GPU

**GPU = Graphics Processing Unit**
- Chip in computer video cards, PlayStation 3, Xbox, etc.
- Two major vendors: NVIDIA and ATI (now AMD)
Enter the GPU

- GPUs are massively multithreaded manycore chips
  - NVIDIA Tesla products have up to 128 scalar processors
  - Over 12,000 concurrent threads in flight
  - Over 470 GFLOPS sustained performance

- Users across science & engineering disciplines are achieving 100x or better speedups on GPUs

- CS researchers can use GPUs as a research platform for manycore computing: arch, PL, numeric, …
GTX Titan: For High Performance Gaming Enthusiasts

CUDA Cores: 2688
Single Precision: ~4.5 Tflops
Double Precision: ~1.27 Tflops
Memory Size: 6GB
Memory B/W: 288GB/s
Heterogeneous Computing

- Terminology:
  - Host: The CPU and its memory (host memory)
  - Device: The GPU and its memory (device memory)
CUDA
Programming Model
total_hits = 0;
sample_points_per_thread = sample_points / num_threads;

for (i = 0; i < num_threads; i++){
    my_arg[i].t_seed = i;  /* can chose any seed – here i is chosen*/
    pthread_create (&p_threads[i], &attr, compute_pi, (void*) &my_arg[i]);
}

for (i = 0; i < num_threads; i++){
    pthread_join (p_threads[i], NULL);
    total_hits += my_arg[i].hits;
}

computed_pi = 4.0*(double) total_hits / ((double) (sample_points));
CUDA Accelerates Computing

Choose the right processor for the right task

CPU
Several sequential cores

CUDA GPU
Thousands of parallel cores
```cpp
#include <iostream>
#include <algorithm>
using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int gindex = threadIdx.x + blockIdx.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;
    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }
    // Synchronize (ensure all the data is available)
    __syncthreads();
    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS; offset <= RADIUS; offset++)
        result += temp[lindex + offset];
    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out; // host copies of a, b, c
    int *d_in, *d_out; // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);
    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);
    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);
    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);
    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE, BLOCK_SIZE>>>(d_in + RADIUS, d_out + RADIUS);
    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

Heterogeneous Computing

**parallel fn**

**serial code**

**parallel code**

**serial code**
1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
CUDA Kernels: Parallel Threads

- A **kernel** is a function executed on the GPU as an array of threads in parallel.
- All threads execute the same code, can take different paths.
- Each thread has an ID:
  - Select input/output data
  - Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels

- Threads are grouped into **blocks**
- **Blocks** are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Kernel Execution

- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store memory that they all use
  - Share results with each other
  - Cooperate to produce a single result
  - Synchronize with each other
Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs
Warps

- Blocks are divided into 32 thread wide units called warps
  - Size of warps is implementation specific and can change in the future

- The SM creates, manages, schedules and executes threads at warp granularity
  - Each warp consists of 32 threads of contiguous threadIds

- All threads in a warp execute the same instruction
  - If threads of a warp diverge the warp serially executes each branch path taken

- When a warp executes an instruction that accesses global memory it coalesces the memory accesses of the threads within the warp into as few transactions as possible
Threads are grouped into **thread blocks**

Kernel = **grid** of thread blocks

---

By definition, threads in the same block may **synchronize with barriers**

```c
scratch[threadID] = begin[threadID];
__syncthreads();
int left = scratch[threadID - 1];
```
Heterogeneous Memory Model

- Host memory
- Device 0 memory
- Device 1 memory

`cudaMemcpy()`
Kernel Memory Access

- **Per-thread**
  - Thread
  - Registers: On-chip
  - Local Memory: Off-chip, uncached

- **Per-block**
  - Block
  - Shared Memory:
    - On-chip, small
    - Fast

- **Per-device**
  - Global Memory:
    - Off-chip, large
    - Uncached
    - Persistent across kernel launches
    - Kernel I/O

© NVIDIA Corporation 2009
Physical Memory Layout

- “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
10-Series Architecture

- 240 thread processors execute kernel threads
- 30 multiprocessors, each contains
  - 8 thread processors
  - One double-precision unit
- Shared memory enables thread cooperation
Execution Model

**Software**
- Threads are executed by thread processors

**Hardware**
- Thread blocks are executed on multiprocessors
- Thread blocks do not migrate
- Several concurrent thread blocks can reside on one multiprocessor - limited by multiprocessor resources (shared memory and register file)
- A kernel is launched as a grid of thread blocks
- Only one kernel can execute on a device at one time

© NVIDIA Corporation 2009
CUDA Programming Basics

Part I - Software Stack and Memory Management
Compiler

- Any source file containing language extensions, like "<<< >>>", must be compiled with `nvcc`
- `nvcc` is a compiler driver
  - Invokes all the necessary tools and compilers like cudacc, g++, cl, ...
- `nvcc` can output either:
  - C code (CPU code)
    - That must then be compiled with the rest of the application using another tool
  - PTX or object code directly
- An executable requires linking to:
  - Runtime library (`cudart`)
  - Core library (`cuda`)
Compiling

CPU/GPU Source

NVCC

PTX Code

Virtual

PTX to Target Compiler

G80

..., GPU

Target code

Physical
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);
```
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);
    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;
}
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);
    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++) 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;
}
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);

    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;
}
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);
    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++) a_h[i] = 100.f + i;
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToDevice);
}

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;
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);
    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++) 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;
}
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;
}
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);
    cudaMemcpy(a_d, a_h, nBytes, cudaMemcpyHostToDevice);
    cudaMemcpy(b_d, a_d, nBytes, cudaMemcpyDeviceToDevice);

    for (i=0, i<N; i++) a_h[i] = 100.f + i;
    cudaMemcpy(b_h, b_d, nBytes, cudaMemcpyDeviceToHost);
    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;
}
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;
}
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;
}
CUDA Programming Basics

Part II - Kernels
Thread Hierarchy

 Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch

 Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
Executing Code on the GPU

- 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
Function Qualifiers

- **__global__**
  - Function called from host and executed on device
  - Must return void

- **__device__**
  - Function called from device and run on device
  - Cannot be called from host code

- **__host__**
  - Function called from host and executed on host (default)
  - **__host__** and **__device__** qualifiers can be combined to generate both CPU and GPU code
Launching Kernels

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
More on Thread and Block IDs

- Threads and blocks have IDs
  - So each thread can 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
Execution Configuration Examples

```
dim3 grid, block;
grid.x = 2; grid.y = 4;
block.x = 8; block.y = 16;
```

```
kernel<<<grid, block>>>(...);
```

```
dim3 grid(2, 4), block(8,16);
```

```
kernel<<<grid, block>>>(...);
```

```
kernel<<<8,1024>>>(...);
```

Equivalent assignment using constructor functions
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
Unique Thread IDs

- Built-in variables are used to determine unique thread IDs
- Map from local thread ID (threadIdx) to a global ID which can be used as array indices

Grid

\[
\text{blockIdx.x} = 0, 1, 2, 3, 4 \\
\text{blockDim.x} = 5 \\
\text{threadIdx.x} = 0, 1, 2, 3, 4
\]

\[
\text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
\]
Minimal Kernels

```c
__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

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

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

```c
__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
Increment Array Example

CPU program

```c
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

```c
__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);
    ...
}
```
Host Synchronization

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel executes after all previous CUDA calls have completed
- `cudaMemcpy()` is synchronous
  - control returns to CPU after copy completes
  - copy starts after all previous CUDA calls have completed
- `cudaThreadSynchronize()`
  - blocks until all previous CUDA calls complete
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);

...
Variable Qualifiers (GPU code)

**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
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
GPU Atomic Integer Operations

- Requires hardware with compute capability >= 1.1
  - G80 = Compute capability 1.0
  - G84/G86/G92 = Compute capability 1.1
  - GT200 = Compute capability 1.3

- Atomic operations on integers in global memory:
  - Associative operations on signed/unsigned ints
  - add, sub, min, max, ...
  - and, or, xor
  - Increment, decrement
  - Exchange, compare and swap

- Atomic operations on integers in shared memory
  - Requires compute capability >= 1.2
Computing $y = ax + y$ with a Serial Loop

```c
void saxpy_serial(int n, float alpha, float *x, float *y)
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Computing $y = ax + y$ in parallel using CUDA

```c
__global__ void saxpy_parallel(int n, float alpha, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i<n ) y[i] = alpha*x[i] + y[i];
}

// Invoke parallel SAXPY kernel (256 threads per block)\
int nbblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
```
Parallel reduction

__global__ void plus_reduce(int *input, int N, int *total)
{
    int tid = threadIdx.x;
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    // Each block loads its elements into shared memory
    __shared__ int x[blocksize];
    x[tid] = input[i];  // assuming that N is a multiple of the block size
    __syncthreads();

    // Build summation tree over elements.
    for(int s=blockDim.x/2; s>0; s=s/2)
    {
        if(tid < s) x[tid] += x[tid + s];
        __syncthreads();
    }

    // Thread 0 adds the partial sum to the total sum
    if( tid == 0 ) atomicAdd(total, x[tid]);
}

Parallel reduction
Coalescing

- Global memory access of 32, 64, or 128-bit words by a half-warp of threads can result in as few as one (or two) transaction(s) if certain access requirements are met.
- Depends on compute capability
  - 1.0 and 1.1 have stricter access requirements.

**Examples – float (32-bit) data**

Global Memory

- 64B aligned segment (16 floats)
- 128B aligned segment (32 floats)

Half-warp of threads
Coalescing
Compute capability 1.0 and 1.1

K-th thread must access k-th word in the segment (or k-th word in 2 contiguous 128B segments for 128-bit words), not all threads need to participate.

Coalesces – 1 transaction

Out of sequence – 16 transactions

Misaligned – 16 transactions
Coalescing
Compute capability 1.2 and higher

- Coalescing is achieved for any pattern of addresses that fits into a segment of size: 32B for 8-bit words, 64B for 16-bit words, 128B for 32- and 64-bit words
- Smaller transactions may be issued to avoid wasted bandwidth due to unused words

1 transaction - 64B segment

2 transactions - 64B and 32B segments

1 transaction - 128B segment
Occupancy

Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy.

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently.

Limited by resource usage:
- Registers
- Shared memory
Blocks per Grid Heuristics

- # of blocks > # of multiprocessors
  - So all multiprocessors have at least one block to execute

- # of blocks / # of multiprocessors > 2
  - Multiple blocks can run concurrently in a multiprocessor
  - Blocks that aren’t waiting at a __syncthreads() keep the hardware busy
  - Subject to resource availability – registers, shared memory

- # of blocks > 100 to scale to future devices
  - Blocks executed in pipeline fashion
  - 1000 blocks per grid will scale across multiple generations
Register Dependency

Read-after-write register dependency

- Instruction’s result can be read ~24 cycles later (assuming 8 cores per SM -- 4 cycles per warp instruction)

Scenarios:

**CUDA:**

```
x = y + 5;
z = x + 3;
s_data[0] += 3;
```

**PTX:**

```
add.f32 $f3, $f1, $f2
add.f32 $f5, $f3, $f4
ld.shared.f32 $f3, [$r31+0]
add.f32 $f3, $f3, $f4
```

To completely hide the latency:

- Run at least 192 threads (6 warps) per multiprocessor
  - At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
- Threads do not have to belong to the same thread block
Register Pressure

- Hide latency by using more threads per multiprocessor

Limiting Factors:

- Number of registers per kernel
  - 8K/16K per multiprocessor, partitioned among concurrent threads

- Amount of shared memory
  - 16KB per multiprocessor, partitioned among concurrent threadblocks

Compile with `-ptxas-options=-v` flag

Use `-maxrregcount=N` flag to NVCC

- N = desired maximum registers / kernel
- At some point “spilling” into local memory may occur
  - Reduces performance – local memory is slow

© NVIDIA Corporation 2009
Optimizing threads per block

Choose threads per block as a multiple of warp size
- Avoid wasting computation on under-populated warps
- Facilitates coalescing

More threads per block != higher occupancy
- Granularity of allocation
  Eg. compute capability 1.1 (max 768 threads/multiprocessor)
    - 512 threads/block => 66% occupancy (can fit only one block)
    - 256 threads/block can have 100% occupancy (can fit 3 blocks)

Heuristics
- Minimum: 64 threads per block
  - Only if multiple concurrent blocks
- 192 or 256 threads a better choice
  - Usually still enough regs to compile and invoke successfully
- This all depends on your computation, so experiment!