What we’ve covered so far…

- What: overview of issues that will come up later on
  - From C code to machine instructions
  - ISA: RISC vs. CISC architectures
  - Microarchitecture issues
  - The transistor, and its roles (CU & SRAM)
  - Registers
  - Pipelining
  - Memory aspects: caches and the virtual memory space
  - Three walls to sequential computing
Flynn’s Taxonomy of Architectures

- There are several ways to classify architectures (based on how memory is organized/accessed, for instance – discussed later)

- Below, classification based on how instructions are executed in relation to data

  - SISD - Single Instruction/Single Data
  - SIMD - Single Instruction/Multiple Data
  - MISD - Multiple Instruction/Single Data
  - MIMD - Multiple Instruction/Multiple Data
Single Instruction/Single Data Architectures

PU – Processing Unit

Your desktop, before the spread of dual core CPUs

Increasing Throughput of SISD

Instructions: 

Pipelining

Multiple Issue
Single Instruction/Multiple Data Architectures

Processors that execute same instruction on multiple pieces of data: NVIDIA GPUs

Each core runs the same set of instructions on different data.

Examples:
- Graphics Processing Unit (GPU): processes pixels of an image in parallel.
- CRAY’s vector processor, see image below.
SISD versus SIMD

Writing a compiler for SIMD architectures is difficult (inter-thread communication complicates the picture...)

Slide Source: ars technica, Peakstream article
Multiple Instruction/Single Data

Not useful, not aware of any commercial implementation...

Almost all our desktop/laptop chips are MIMD systems
Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories

Instructions: 🔄 △ + □
Amdahl’s Law


“A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude”

- Let \( r_s \) capture the amount of time that a program spends in components that can only be run sequentially
- Let \( r_p \) capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that \( r_s \) and \( r_p \) are normalized, so that \( r_s + r_p = 1 \)
- Let \( n \) be the number of threads used to parallelize the part of the program that can be executed in parallel
- The “best case scenario” speedup \( S \) is

\[
S = \frac{T_{old}}{T_{new}} = \frac{r_s + r_p}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}
\]
Amdahl’s Law
[Cntd.]

- Sometimes called the law of diminishing returns

- In the context of parallel computing used to illustrate how going parallel with a part of your code is going to lead to overall speedups

- The art is to find for the same problem an algorithm that has a large $r_p$
  - Sometimes requires a completely different angle of approach for a solution

- Nomenclature
  - Algorithms for which $r_p=1$ are called “embarrassingly parallel”
Example: Amdahl's Law

- Suppose that a program spends 60% of its time in I/O operations, pre and post-processing.
- The rest of 40% is spent on computation, most of which can be parallelized.
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem. What is the maximum amount of speedup that you can expect given this investment?
- Asymptotically, what is the maximum speedup that you can ever hope for?
A Word on “Scaling”
[important to understand]

- **Algorithmic Scaling** of a solution algorithm
  - You only have a mathematical solution algorithm at this point
  - Refers to how the effort required by the solution algorithm scales with the size of the problem
  - Examples:
    - Naïve implementation of the N-body problem scales like $O(N^2)$, where $N$ is the number of bodies
    - Sophisticated algorithms scale like $O(N \log N)$
    - Gauss elimination scales like the cube of the number of unknowns in your linear system

- **Implementation Scaling** of a solution on a certain architecture
  - **Intrinsic Scaling**: how the execution time changes with an increase in the size of the problem
  - **Strong Scaling**: how the execution time changes when you increase the processing resources
  - **Weak Scaling**: how the execution time changes when you increase the problem size but also the processing resources in a way that basically keeps the ratio of problem size/processor constant
A Word on “Scaling”
[important to understand]

- Two follow up comments

1. Worry about this: Is the Intrinsic Scaling similar to the Algorithmic Scaling?
   - If Intrinsic Scaling significantly worse than Algorithmic Scaling you then probably memory transactions are dominating the implementation

2. If the problem doesn’t scale can be an interplay of several factors
   - The intrinsic nature of the problem at hand
   - The attributes (organization) of the underlying hardware
   - The algorithm used to solve the problem; i.e., the amount of parallelism it exposes
End: Intro Part

Beginning: GPU Computing, CUDA Programming Model
Layout of Typical Hardware Architecture

GPU w/ local DRAM (the “device”)
Intel Haswell

- June 2013
- 22 nm technology
- 1.4 billion transistors
- 4 cores, hyperthreaded
- Integrated GPU
- System-on-a-chip (SoC) design
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers

- Kepler K20X delivers 1.515 Tflops in double precision

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - “Extended C”
Fermi: 30,000 Feet Perspective

- Lots of ALU (green), not much of CU (orange)
- Explains why GPUs are fast for high arithmetic intensity applications
- Arithmetic intensity: high when many operations performed per word of memory
The Fermi Architecture

- Early 2010
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SP, “shaders”)
- 64 KB L1 cache
- 768 KB L2 uniform cache (shared by all SMs)
- Up to 6 GB of global memory
- Operates at several clock rates
  - Memory
  - Scheduler
  - Shader (SP)
- High memory bandwidth
  - Close to 200 GB/s
## Key Parameters

### GPU, CPU

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processing Cores</strong></td>
<td>448</td>
<td>4 (8 threads)</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>- 48 KB L1/SM</td>
<td>- 32 KB L1 cache / core</td>
</tr>
<tr>
<td></td>
<td>- 768 KB (all SMs)</td>
<td>- 256 KB L2 (I&amp;D)cache / core</td>
</tr>
<tr>
<td></td>
<td>- No L3 cache</td>
<td>- 8 MB L3 (I&amp;D) shared by all cores</td>
</tr>
<tr>
<td></td>
<td>3 GB (global mem)</td>
<td>Can have lots of main memory</td>
</tr>
<tr>
<td><strong>Clock speed</strong></td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td><strong>Memory bandwidth</strong></td>
<td>140 GB/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td><strong>Floating point operations/s</strong></td>
<td>515 x 10^9 Double Precision</td>
<td>70 x 10^9 Double Precision</td>
</tr>
</tbody>
</table>
Why is the GPU Fast?

- The GPU is specialized for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
  - More transistors devoted to data processing rather than data caching and control flow
  - Where are GPUs good: high arithmetic intensity (the ratio between arithmetic operations and memory operations)

- The large video game industry exerts strong economic pressure that forces constant innovation in GPU computing
Bandwidth in a CPU-GPU System

NOTE: The width of the black lines is proportional to the bandwidth.

IMPORTANT: this is an important slide...
Recall the **Very Important Point**
[made yesterday]

- Almost all our applications are bound by memory speed
  - Most often, the cores (or SPs) idle waiting for data
  - What’s important is how fast you can move data
    - You want high memory bandwidth
High Speed Requires High Bandwidth

- Assume that you want to add two arrays and reach 1 Tflops: $1 \times 10^{12}$ ops/second
- You need to fetch/feed $2 \times 10^{12}$ operands per second…
- Assume each number is stored using 4 bytes (float or integer)
- You need to fetch $2 \times 10^{12} \times 4$ bytes in a second. This is $8 \times 10^{12}$ B/s, which is 8 TB/s…
- You haven’t taken into account that you probably want to send back the outcome of the operation that you carry out

- The global memory bandwidth on the GPU is in the neighborhood of 0.2 TB/s
  - About 40 times less than what you need
When Are GPUs Good?

- Ideally suited for data-parallel computing (SIMD)

- Moreover, you want to have high arithmetic intensity
  - Arithmetic intensity: ratio or arithmetic operations to memory operations

- You are off to a good start with GPU computing if you can do this…
  - GET THE DATA ON THE GPU AND KEEP IT THERE
  - GIVE THE GPU ENOUGH WORK TO DO
  - FOCUS ON DATA REUSE WITHIN THE GPU TO AVOID MEMORY BANDWIDTH LIMITATIONS
GPU Computing – The Basic Idea

- GPU, going beyond graphics:
  - The GPU is connected to the CPU by a reasonable fast bus
    - PCI 3.0 - 12 GB/s is typical today
      - That is, 12 GB/s in each direction, and you can move data simultaneously in both directions
  - The idea is to use the GPU as a co-processor
    - Farm out big parallel jobs to the GPU
    - CPU stays busy with the execution of “corner” tasks
    - You have to copy data down into the GPU, and then fetch results back
      - Ok if this data transfer is overshadowed by the number crunching done using that data (remember Amdahl’s law…)
CUDA: Making the GPU Tick…

- “Compute Unified Device Architecture” – freely distributed by NVIDIA
- When introduced it eliminated the constraints associated with GPGPU
- It enables a general purpose programming model
  - User kicks off batches of threads on the GPU to execute a function (kernel)
- Targeted software stack
  - Scientific computing oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Interface designed for compute, graphics-free API
  - Explicit GPU memory management
CUDA Programming Model: GPU as a Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (global memory in CUDA parlance)
  - Runs many threads in parallel

- Data-parallel portions of an application run on the device as kernels which are executed in parallel by 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 heavy ones
GPU Processor Terminology

- GPU is a SIMD device → it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that the GPU is assigned to process
  - The NVIDIA calls this model SIMT (single instruction multiple thread)

- The number crunching power comes from a vertical hierarchy:
  - A collection of Streaming Multiprocessors (SMs)
  - Each SM has a set of 32 Scalar Processors (SPs)
    - Maxwell has 128 SPs, Kepler has 196 SPs, Fermi 2.1 had 48 SPs

- The quantum of scalability is the SM
  - The more $ you pay, the more SMs you get inside your GPU
  - Fermi can have up to 16 SMs on one GPU card
"Compute Capability of a Device" refers to hardware
- Defined by a major revision number and a minor revision number

Example:
- Tesla C1060 is compute capability 1.3
- Fermi architecture is capability 2
- Kepler architecture is capability 3 (on Euler now)
- Maxwell architecture is capability 4 (no card on Euler)
- Pascal architecture is capability 5 (on Euler now)
- The minor revision number indicates incremental changes within an architecture class

A higher compute capability indicates an more able piece of hardware

The "CUDA Version" indicates what version of the software you are using to run on the hardware
- Right now, the most recent version of CUDA is 8.0

In a perfect world
- You would run the most recent CUDA (version 8.0) software release
- You would use the most recent architecture (compute capability 5.X)
Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - Makes sense, the functionality in later versions increased, was not there in previous versions
## GPU Computing Applications

### Libraries and Middleware
- CUFFT
- CUBLAS
- CURAND
- CURSPARSE
- CULA
- MAGMA
- Thrust
- NPP
- VSIPL
- SVM
- OpenCurrent
- PhysX
- OptiX
- iray
- MATLAB
- Mathematica

### Programming Languages
- C
- C++
- Fortran
- Java
- Python Wrappers
- DirectCompute
- Directives (e.g. OpenACC)

### CUDA-Enabled NVIDIA GPUs

<table>
<thead>
<tr>
<th>Architecture</th>
<th>GeForce</th>
<th>Quadro</th>
<th>Tesla</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kepler Architecture</td>
<td>600 Series</td>
<td>Kepler Series</td>
<td>K20, K10</td>
</tr>
<tr>
<td>Fermi Architecture</td>
<td>500 Series,</td>
<td>Fermi Series</td>
<td>20 Series</td>
</tr>
<tr>
<td></td>
<td>400 Series</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tesla Architecture</td>
<td>200 Series,</td>
<td>FX Series,</td>
<td>10 Series</td>
</tr>
<tr>
<td></td>
<td>9 Series,</td>
<td>Plex Series,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 Series</td>
<td>NVS Series,</td>
<td></td>
</tr>
</tbody>
</table>

### Applications
- Entertainment
- Professional Graphics
- High Performance Computing
The CUDA Execution Model
GPU Computing – The Basic Idea

- The GPU is linked to the CPU by a reasonably fast connection

- The idea is to use the GPU as a co-processor
  - Farm out big parallel tasks to the GPU
  - Keep the CPU busy with the control of the execution and “corner” tasks
The CUDA Execution Model is Asynchronous

This is how your C code looks like…

This is how the code gets executed on the hardware in heterogeneous computing. GPU calls are asynchronous…
Languages Supported in CUDA

- Note that everything is done in C
  - Yet minor extensions are needed to flag the fact that a function actually represents a kernel, that there are functions that will only run on the device, etc.
    - You end up working in “C with extensions”
  - FORTRAN is supported
  - There is [limited] support for C++ programming (operator overload, for instance)
CUDA Function Declarations
(the “C with extensions” part)

<table>
<thead>
<tr>
<th></th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong></td>
<td>float myDeviceFunc()</td>
<td>device</td>
</tr>
<tr>
<td><strong>global</strong></td>
<td>void myKernelFunc()</td>
<td>device</td>
</tr>
<tr>
<td><strong>host</strong></td>
<td>float myHostFunc()</td>
<td>host</td>
</tr>
</tbody>
</table>

- __global__ defines a kernel function, launched by host, executed on the device
  - Must return void
- For a full list, see CUDA Reference Manual
#include <cutil_inline.h>
#include <iostream>

__global__ void simpleKernel(int* data)
{
    //write something trivial to the global memory...
    data[threadIdx.x] = blockIdx.x + threadIdx.x;
}

int main()
{
    int hostArray[4], *devArray;
    //allocate memory on the device (GPU)
    cudaMalloc((void**)&devArray, sizeof(int)*4);

    //invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,4>>>(devArray);

    //bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int)*4, cudaMemcpyDeviceToHost);

    //print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: 
";
    std::cout << hostArray[0] << ", ";
    std::cout << hostArray[1] << ", ";
    std::cout << hostArray[2] << ", ";
    std::cout << hostArray[3] << std::endl;

    //release the memory allocated on the GPU
    cudaFree(devArray);

    return 0;
}
CUDA, Second Example

- Multiply, pairwise, two arrays of 3 million integers

```c
int main(int argc, char* argv[])
{
    const int arraySize = 3000000;
    int *hA, *hB, *hC;
    setupHost(&hA, &hB, &hC, arraySize);

    int *dA, *dB, *dC;
    setupDevice(&dA, &dB, &dC, arraySize);

    cudaMemcpy(dA, hA, sizeof(int) * arraySize, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, hB, sizeof(int) * arraySize, cudaMemcpyHostToDevice);

    const int threadsPerBlock = 512;
    const int blockSizeMultiplication = arraySize/threadsPerBlock + 1;
    multiply_ab<<<blockSizeMultiplication,threadsPerBlock>>>(dA, dB, dC, arraySize);
    cudaMemcpy(hC, dC, sizeof(int) * arraySize, cudaMemcpyDeviceToHost);

    cleanupHost(hA, hB, hC);
    cleanupDevice(dA, dB, dC);
    return 0;
}
```
CUDA, Second Example

[Cntd.]

```c
__global__ void multiply_ab(int* a, int* b, int* c, int size)
{
    int whichEntry = threadIdx.x + blockIdx.x*blockDim.x;
    if( whichEntry<size )
        c[whichEntry] = a[whichEntry]*b[whichEntry];
}

void setupDevice(int** pdA, int** pdB, int** pdC, int arraySize)
{
    cudaMalloc((void**) pdA, sizeof(int) * arraySize);
    cudaMalloc((void**) pdB, sizeof(int) * arraySize);
    cudaMalloc((void**) pdC, sizeof(int) * arraySize);
}

void cleanupDevice(int *dA, int *dB, int *dC)
{
    cudaFree(dA);
    cudaFree(dB);
    cudaFree(dC);
}
```
Review of Nomenclature

- **The HOST**
  - This is your CPU executing the “master” thread

- **The DEVICE**
  - This is the GPU card, connected to the HOST through a PCIe X16 connection

- **The HOST (the master CPU thread) calls DEVICE to execute KERNEL**

- **When calling the KERNEL, the HOST also has to inform the DEVICE how many threads should each execute the KERNEL**
  - This is called “defining the execution configuration”
The Concept of Execution Configuration

- A kernel function must be called with an execution configuration:

  ```
  __global__ void kernelFoo(...); // declaration
  dim3 DimGrid(100, 50);          // 5000 thread blocks
  dim3 DimBlock(4, 8, 8);          // 256 threads per block
  kernelFoo<<< DimGrid, DimBlock>>>(...your arg list comes here…);
  ```

- Recall that any call to a kernel function is **asynchronous**
  - By default, execution on host doesn’t wait for kernel to finish
The host call below instructs the GPU to execute the function (kernel) “foo” using 25,600 threads
- Two arguments are passed down to each thread executing the kernel “foo”

```
foo<<<100,256>>>(pMyMatrixD, pMyVecD)
```

- In this execution configuration, the host instructs the device that it is supposed to run 100 blocks each having 256 threads in it
- The concept of block is important since it represents the entity that gets executed by an SM (stream multiprocessor)
More on the Execution Model

[Some Constraints]

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of 65,535 by 65,535 by 65,535 grid of blocks (about 280,000 billion blocks)

- Threads in each block:
  - The threads can be organized as a 3D structure (x,y,z)
  - The total number of threads in each block cannot be larger than 1024
Matrix Multiplication Example
Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Use only global memory (don’t bring shared memory into picture yet)
  - Matrix will be of small dimension, job can be done using one block
  - Concentrate on
    - Thread ID usage
  - Memory data transfer API between host and device
Matrix Data Structure

The following data structure will come in handy

- Purpose: store matrix-related data
- Note that the matrix is stored in row-major order in a one dimensional array pointed to by “elements”

```c
// IMPORTANT - Matrices are stored in row-major order:
// M(row, col) = M.elements[row * M.width + col]

typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $\text{WIDTH} \times \text{WIDTH}$
  - Assume $\text{WIDTH}$ was defined to be 32

- Software Design Decisions:
  - One thread handles one element of $P$
  - Each thread will access all the entries in one row of $M$ and one column of $N$
    - 2*WIDTH read accesses to global memory
    - One write access to global memory
Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes **one** element of P

- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good…

- Size of matrix limited by the number of threads allowed in a thread block
Matrix Multiplication:
Sequential Approach, Coded in C

// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; //march along a row of M
                double b = N.elements[k * N.width + j]; //march along a column of N
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
Step 1: Matrix Multiplication, Host-side.
Main Program Code

```c
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
```
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P) {
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

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

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
Step 4: Matrix Multiplication - Device-side Kernel Function

```c
// Matrix multiplication kernel - thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P) {
    // 2D Thread Index; computing P[ty][tx]...
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx].
    // That is, P.elements[ty * P.width + tx] = Pvalue
    float Pvalue = 0;

    for (int k = 0; k < M.width; ++k) {
        float Melement = M.elements[ty * M.width + k];
        float Nelement = N.elements[k * N.width + tx];
        Pvalue += Melement * Nelement;
    }

    // Write matrix to device memory; each thread one element
    P.elements[ty * P.width + tx] = Pvalue;
}
```

Step 4: Some Loose Ends

// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMemcpy((void**)&Mdevice.elements, Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size, cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size, cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
Threads and blocks have indices
- Used by each thread to decide what data to work on (more later)
- Block Index: a triplet of uint
- Thread Index: a triplet of uint

Why this 3D layout?
- Simplifies memory addressing when processing multidimensional data
  - Handling matrices
  - Solving PDEs on 3D subdomains
  - ...
A Couple of Built-In Variables
[Critical in supporting the SIMD parallel computing paradigm]

- It’s essential for each thread to be able to find out the grid and block dimensions and its block index and thread index.

- Each thread when executing a kernel has access to the following read-only built-in variables:
  - `threadIdx (uint3)` – contains the thread index within a block
  - `blockDim (dim3)` – contains the dimension of the block
  - `blockIdx (uint3)` – contains the block index within the grid
  - `gridDim (dim3)` – contains the dimension of the grid
  - `[warpSize (uint) – provides warp size, we’ll talk about this later…]`
Thread Index vs. Thread ID
[critical in (i) understanding how SIMD is supported in CUDA, and (ii) understanding the concept of “warp”]

- Each block organizes its threads in a 3D structure defined by its three dimensions: $D_x$, $D_y$, and $D_z$ that you specify.

- A block cannot have more than 1024 threads $\Rightarrow D_x \times D_y \times D_z \leq 1024$.

- Each thread in a block can be identified by a unique index $(x, y, z)$, and

$$0 \leq x < D_x \quad 0 \leq y < D_y \quad 0 \leq z < D_z$$

- A triplet $(x, y, z)$, called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as $t_{id} = x + y \times D_x + z \times D_x \times D_y$. You can regard this as a ”projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on ”warps” later).

- In general, operating for vectors typically results in you choosing $D_y = D_z = 1$. Handling matrices typically goes well with $D_z = 1$. For handling PDEs in 3D you might want to have all three block dimensions nonzero.
Check your understanding

- How was the grid defined for this pic?  
  - I.e., how many blocks in X and Y directions?

- How was a block defined in this pic?
Example: Array Indexing

- Purpose of Example: see a scenario of how multiple blocks are used to index entries in an array

- First, recall this: there is a limit on the number of threads you can squeeze in a block – you can have up to 1024 threads

- Note: In the vast majority of applications you need to use many blocks (each of which contains the same number N of threads) to get a job done. Next example puts things in perspective
Example: Array Indexing
[Important to grasp: shows thread to task mapping]

- No longer as simple as using only `threadIdx.x`
  - Consider indexing into an array, one thread accessing one element
  - Assume you have $M=8$ threads per block and the array is 32 entries long

- With $M$ threads per block a unique index for each thread is given by:
  
  $$\text{int index} = \text{threadIdx.x} + \text{blockIdx.x} \times M;$$

[NVIDIA]→
Example: Array Indexing

- What will be the array entry that thread of index 5 in block of index 2 will work on?

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Imagine you are one of many threads, and you have your thread index and block index

- You need to figure out what the work you need to do is
  - Just like we did on previous slide, where thread 5 in block 2 mapped into 21

- You have to make sure you actually need to do that work
  - In many problems there are threads, typically of large id, that need to do no work
  - Example: you launch two blocks with 512 threads but your array is only 1000 elements long. Then 24 threads at the end do nothing
Before Moving On…
[Some Words of Wisdom]

● In GPU computing you use as many threads as data items (tasks, jobs) you have to perform
  ● This replaces the purpose in life of the “for” loop
  ● Number of threads & blocks is established at run-time

● Number of threads = Number of data items (tasks)
  ● It means that you’ll have to come up with a rule to match a thread to a data item (task) that this thread needs to process
  ● Most common source of errors and frustration in GPU computing
    ● It never fails to deliver (frustration)
      :-(
Timing Your Application

- Timing support – part of the CUDA API
  - You pick it up as soon as you include `<cuda.h>`

- Why it is good to use
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

- Reports time in milliseconds, accurate within 0.5 microseconds
  - From NVIDIA CUDA Library Documentation:
    - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`. If either event has been recorded with a non-zero stream, the result is undefined.
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);

    cudaEventRecord(startEvent, 0);

    yourKernelCallHere<<<NumBlk,NumThrds>>>(args);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);
    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";

    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
Execution Scheduling Issues
[NVIDIA cards specific]
Thread Execution Scheduling

- Topic we are about to discuss:
  - You launch on the device many blocks, each containing many threads
  - Several blocks can get executed simultaneously on one SM. How is this possible?
The 30,000 Feet Perspective

- There are two schedulers at work in GPU computing
  - A device-level scheduler: assigns blocks to SM that indicate at a given time “excess capacity”
  - An SM-level scheduler, which schedules the execution of the threads in a block onto the functional units available to an SM
  - The more interesting is the SM-level scheduler
Device-Level Scheduler

- Grid is launched on the device

- Thread Blocks are distributed to the SMs
  - Potentially more than one block per SM
  - There is a limit on the number of blocks an SM can take.

- As Thread Blocks complete kernel execution, resources are freed
  - Device-level scheduler can launch next Block[s] in line

- This is the first levels of scheduling:
  - For running [desirably] a large number of blocks on a relatively small number of SMs (16/14/etc.)

- Limits for resident blocks:
  - 32 blocks on Maxwell SMX
  - 16 blocks can be resident on a Kepler SM
  - 8 blocks can be resident on a Fermi & Tesla SM
SM-Level Scheduler[s]

- Each Thread Block divided in 32-thread “warps”
  - “32”: selected by NVIDIA, programmer has no say

- Warps are the basic scheduling unit on the SM

- Limits, number of resident warps on an SM:
  - 64: on Kepler & Maxwell (i.e., 2048 resident threads)
  - 48: on Fermi (i.e., 1536 resident threads)
  - 32: on Tesla (i.e., 1024 resident threads)

- EXAMPLE: If 3 blocks are processed by an SM and each Block has 256 threads, how many warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only one of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements almost zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute same instruction

Cycles needed to dispatch the same instruction for all threads in a warp
- On Tesla: 4 cycles
- On Fermi: 1 cycle
- Recent architectures: more than one instruction/cycle

How is this relevant?
- Suppose you use a Fermi card AND your code has 1 global memory access every 12 simple instructions
- Then, a minimum of 34 Warps are needed to fully tolerate 400-cycle memory latency:
  \[
  \frac{400}{12} = 33.3333 \rightarrow 34 \text{ Warps}
  \]
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads
- One warp issued at each clock cycle by each scheduler
- During no cycle can more than 2 warps be dispatched for execution on the four functional units
- Scoreboarding is used to figure out which warp is ready
Quick Remarks

- Fermi architecture is pipelined
  - Each clock cycle can wrap up one instruction

- Fermi architecture is not superscalar
  - There is no multiple issue that can come into play
Example: Fermi Related

- Scheduler works at 607 MHz
- Functional units work at 1215 MHz

Question:
- What is the peak flop rate of GTX480?
  - 15 SMs * 32 SPs * 1215 * 2 (Fused Multiplied Add) = 1166400 Mflops
  - That is, 1.166 Tflops, single precision
Fermi Specifics

- As illustrated in the picture, at no time can we see more than 2 warps being dispatched for execution during a cycle.
- Note that at any given time we might have more than two functional units working though (which is actually very good, device kept busy).
# NVIDIA GPUs: SM Architecture Specifications

<table>
<thead>
<tr>
<th>Architecture specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Number of ALU lanes for integer and floating-point arithmetic operations</td>
<td>8[19]</td>
</tr>
<tr>
<td>Number of special function units for single-precision floating-point transcendental functions</td>
<td>2</td>
</tr>
<tr>
<td>Number of texture filtering units for every texture address unit or render output unit (ROP)</td>
<td>2</td>
</tr>
<tr>
<td>Number of warp schedulers</td>
<td>1</td>
</tr>
<tr>
<td>Number of instructions issued at once by scheduler</td>
<td>1</td>
</tr>
</tbody>
</table>

References:
1. [NVIDIA](https://developer.nvidia.com/)
2. [NVIDIA](https://developer.nvidia.com/)
3. [NVIDIA](https://developer.nvidia.com/)
4. [NVIDIA](https://developer.nvidia.com/)
5. [NVIDIA](https://developer.nvidia.com/)
6. [NVIDIA](https://developer.nvidia.com/)
7. [NVIDIA](https://developer.nvidia.com/)
8. [NVIDIA](https://developer.nvidia.com/)
9. [NVIDIA](https://developer.nvidia.com/)
10. [NVIDIA](https://developer.nvidia.com/)
Test Your Understanding

• Is the Kepler architecture superscalar?
## Technical Specifications and Features

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Maximum dimensionality of grid of thread blocks</td>
<td>2</td>
</tr>
<tr>
<td>Maximum x-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum y-, or z-dimension of a grid of thread blocks</td>
<td>65535</td>
</tr>
<tr>
<td>Maximum dimensionality of thread block</td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
</tr>
<tr>
<td>Maximum number of 32-bit registers per thread</td>
<td>128</td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
</tr>
<tr>
<td>Constant memory size</td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
</tr>
</tbody>
</table>

[Wikipedia]→
Threads are Executed in Warps

- Each thread block split into one or more warps
- When the thread block size is not a multiple of the warp size, unused threads within the last warp are disabled automatically
- The hardware schedules each warp independently
- Warps within a thread block execute independently

NVIDIA [J. Balfour]→
Organizing Threads into Warps

- Thread IDs within a warp are consecutive and increasing
  - This goes back to the 1D projection from thread index to thread ID
  - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Threads with ID 0 through 31 make up Warp 0, 32 through 63 make up Warp 1, etc.

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps since there is no such thing
  - Warp scheduling is not something the user can control in CUDA
Thread and Warp Scheduling

- An SM can switch between warps with no apparent overhead.
- Warps with instructions whose inputs are ready are eligible to execute, and will be considered when scheduling.
- When a warp is selected for execution, all [active] threads execute the same instruction in lockstep fashion.
Revisiting the Concept of Execution Configuration

- Prefer thread block sizes that result in mostly full warps
  
  **Bad:** \texttt{kernel<<<N, 1>>>(...)}
  
  **Okay:** \texttt{kernel<<<(N+31)/32, 32>>>(...)}
  
  **Better:** \texttt{kernel<<<(N+127)/128, 128>>>(...)}

- Have enough threads per block to provide hardware with many warps to switch between
  
  - This is how the GPU hides memory access latency

- Resource like \texttt{__shared__} may constrain number of threads per block

- Algorithm and decomposition of problem will reveal the preferred amount of shared data and \texttt{__shared__} allocation
  
  - We often have to take a step back and come up with a new algorithm that exposes parallelism
Scheduling: Summing It Up…

- When host invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity

- Up to 8 blocks (on Fermi) can be executed at the same time by an SM
  - Up to 16 on Kepler, 32 on Maxwell

- When a block of threads is executed on an SM, its threads are grouped in warps. The SM manages several warps at the same time
  - Up to 64 warps can be managed on Kepler and Maxwell

- When a thread block finishes, a new block is launched on the vacated SM
Thread Divergence

Consider the following code:

```c
__global__ void odd_even(int n, int* x)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if( (i & 0x01) == 0 )
    {
        x[i] = x[i] + 1;
    }
    else
    {
        x[i] = x[i] + 2;
    }
}
```

Half the threads (even i) in the warp execute the `if` clause, the other half (odd i) the `else` clause.
Thread Divergence

The system automatically handles control flow divergence, conditions in which threads within a warp execute different paths through a kernel.

Often, this requires that the hardware execute multiple paths through a kernel for a warp.
  - For example, both the if clause and the corresponding else clause.
__global__ void kv(int* x, int* y)
{
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int t;
    bool b = f(x[i]);
    if( b )
    {
        // g(x)
        t = g(x[i]);
    }
    else
    {
        // h(x)
        t = h(x[i]);
    }
    y[i] = t;
}
Thread Divergence

[4/4]

- Nested branches are handled similarly
  - Deeper nesting results in more threads being temporarily disabled

- In general, one does not need to consider divergence when reasoning about the correctness of a program
  - Certain code constructs, such as those involving schemes in which threads within a warp spin-wait on a lock, can cause deadlock

- In general, one does need to consider divergence when reasoning about the performance of a program

- NVIDIA calls execution model SIMT (Single Instruction Multiple Threads) to differentiate from actual SIMD where threads really are in lockstep

NVIDIA [J. Balfour]→
Performance of Divergent Code

- Performance decreases with degree of divergence in warps
- Here’s an extreme example…

```c
__global__ void dv(int* x) {
  int i = threadIdx.x + blockDim.x * blockIdx.x;
  switch (i % 32) {
    case 0 : x[i] = a(x[i]); break;
    case 1 : x[i] = b(x[i]); break;
    ...  
    case 31: x[i] = v(x[i]); break;
  }
}
```
Performance of Divergent Code

Compiler and hardware can detect when all threads in a warp branch in the same direction

- Example: all take the if clause, or all take the else clause
- The hardware is optimized to handle these cases without loss of performance

```c
if (threadIdx.x / WARP_SIZE >= 2) {
}
```

- Creates two different control paths for threads in a block
- Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path. There is no warp divergence...

The compiler can also compile short conditional clauses to use predicates (bits that conditional convert instructions into null ops)

- Avoids some branch divergence overheads, and is more efficient
- Often acceptable performance with short conditional clauses