# CSE 746 - Parallel and High Performance Computing Lecture 1 - Introduction to GPU programming

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo
**ppomorsk@sharcnet.ca**
**http://ppomorsk.sharcnet.ca/**

# CSE 746 - Advanced Parallel and High Performance Computing

- Instructor: Dr. Pawel Pomorski ( ppomorsk@sharcnet.ca )
- Course website: http://ppomorsk.sharcnet.ca/CSE_746.html
- Office: E6-2020 at University of Waterloo
- Office hours: please arrange meeting before/after lecture via email
- Lectures: Wednesday 2:30-5:30 in HH 207
- 12 lectures total, no lecture during winter break
- no course materials required
- Students need SHARCNET account, and should bring laptop to class for hands-on activities

# CSE 746 - Advanced Parallel and High Performance Computing

- Evaluation:
  Final project: 40%
  Two assignments: 20% each
  Two in-class quizzes: 10% each
- Final project can be chosen by student in his/her area of interest or research. If that's not a good option, a topic can be selected in consultation with instructor.

# Overview

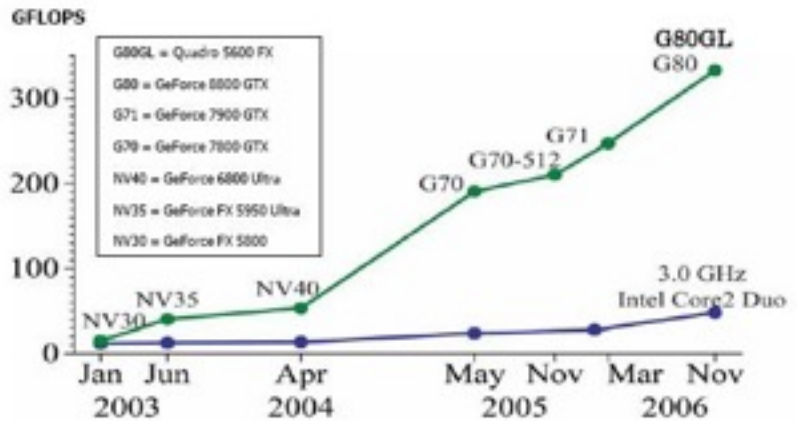
- Introduction to GPU programming
- Introduction to CUDA
- CUDA example programs
- CUDA libraries
- OpenACC
- CUDA extensions to the C programming language
- Beyond the basics - initial discussion on optimizing CUDA

# #1 system on Fall 2012 TOP500 list - Titan


Oak Ridge National Labs - operational in October 2012
18,688 Opteron 16-core CPUs
18,688 NVIDIA Tesla K20 **GPUs**
17.6 peta FLOPS

Fell to #2 on Nov. 2013 list, beat by Intel Phi system

# GPU computing timeline




before 2003 - Calculations on GPU, using graphics API
2003 - Brook “C with streams”
2005 - Steady increase in CPU clock speed comes to a halt, switch to multicore chips to compensate. At the same time, computational power of GPUs increases
November, 2006 - CUDA released by NVIDIA
November, 2006 - CTM (Close to Metal) from ATI
December 2007 - Succeeded by AMD Stream SDK
December, 2008 - Technical specification for OpenCL1.0 released
April, 2009 - First OpenCL 1.0 GPU drivers released by NVIDIA
August, 2009 - Mac OS X 10.6 Snow Leopard released, with OpenCL 1.0 included
September 2009 - Public release of OpenCL by NVIDIA
December 2009 - AMD release of ATI Stream SDK 2.0 with OpenCL support
March 2010 - CUDA 3.0 released, incorporating OpenCL
May 2011 - CUDA 4.0 released, better multi-GPU support
mid-2012 - CUDA 5.0
late-2012 - NVIDIA K20 Kepler cards
Future - CPUs will have so many cores they will start to be treated as GPUs?
Accelerators become universal?

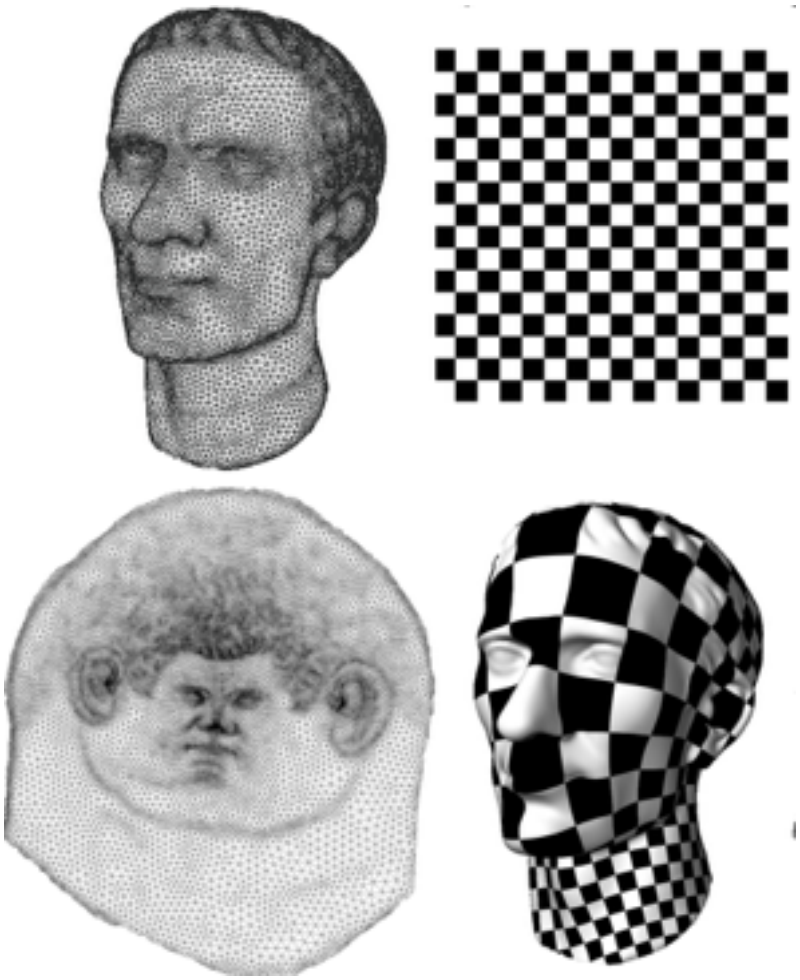
# Introduction to GPU programming

- A graphics processing unit (GPU) is a processor whose main job is to accelerate the rendering of 3D graphics primitives. Performance gains were mostly high performance computer gaming market

- GPU makers have realized that with relatively little additional silicon a GPU can be made into a general purpose computer. They have added this functionality to increase the appeal of cards.
- Even computer games now increasingly take advantage of general compute for game physics simulation

# A brief tour of graphics programming

- 2D textures are wrapped around 3D meshes to assign colour to individual pixels on screen
- Lighting and shadow are applied to bring out 3D features
- Shaders allow programmers to define custom shadow and lighting techniques
  - can also combine multiple textures in interesting ways
- Resulting pixels get sent to a frame buffer for display on the monitor



# Introduction to GPGPU (cont.)

- This was cool for a while, but because all computations occurred within the graphics pipeline, there were limitations:
  - limited inputs/outputs
  - limited data types, graphics-specific semantics
  - memory and processor optimized for short vectors, 2D textures
  - lack of communication or synchronization between “threads”
  - no writes to random memory locations (scatter)
  - no in-out textures (had to ping-pong in multi-pass algorithms)
  - graphics API overhead
  - graphics API learning curve

# General computing APIs for GPUs

- NVIDIA offers ***CUDA*** while AMD has moved toward ***OpenCL*** (also supported by NVIDIA)

- These computing platforms bypass the graphics pipeline and expose the raw computational capabilities of the hardware. Programmer needs to know nothing about graphics programming.

- **OpenACC** compiler directive approach is emerging as an alternative (works somewhat like OpenMP)

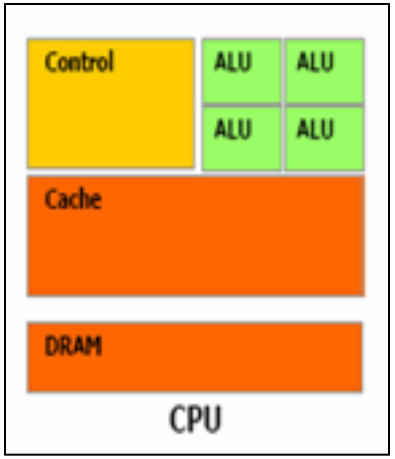
- More recent and less developed alternative to CUDA: ***OpenCL***
  - a vendor-agnostic computing platform
  - supports vendor-specific extensions akin to OpenGL
  - goal is to support a range of hardware architectures including GPUs, CPUs, Cell processors, Larrabee and DSPs using a standard low-level API

# The appeal of GPGPU

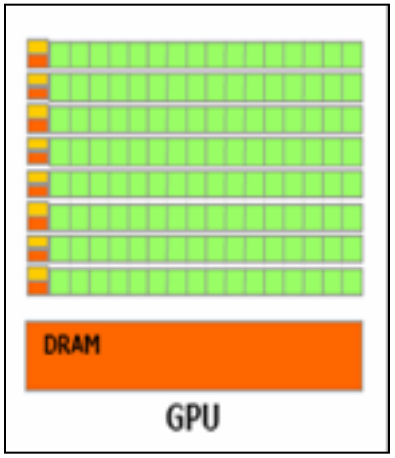
- "Supercomputing for the masses"
  - significant computational horsepower at an attractive price point
  - readily accessible hardware
- Scalability
  - programs can execute without modification on a run-of-the-mill PC with a $150 graphics card or a dedicated multi-card supercomputer worth thousands of dollars
- Bright future – the computational capability of GPUs doubles each year
  - more thread processors, faster clocks, faster DRAM, …
  - "GPUs are getting faster, faster"

# Comparing GPUs and CPUs

- CPU
  - "Jack of all trades"
  - task parallelism (diverse tasks)
  - minimize latency
  - multithreaded
  - some SIMD

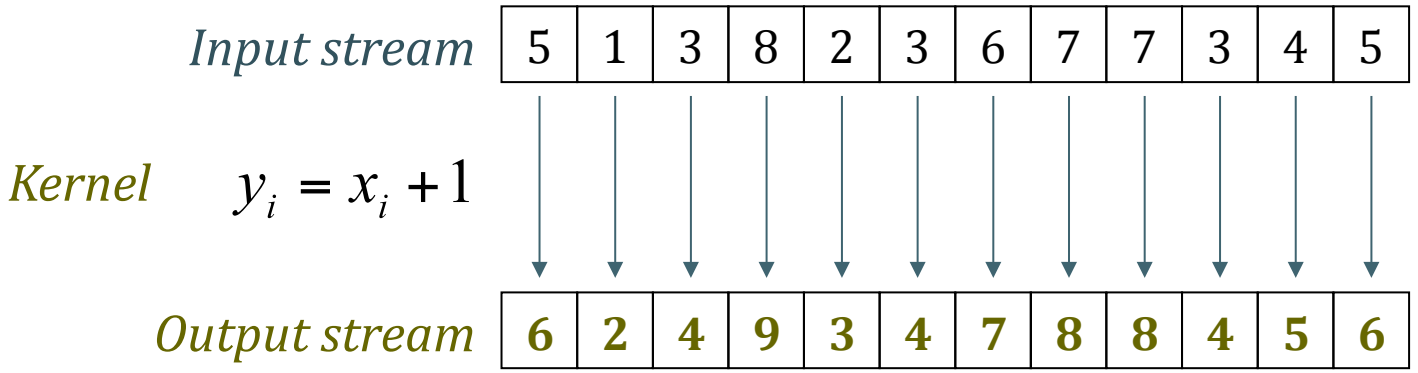



- GPU
  - excel at number crunching
  - data parallelism (single task)
  - maximize throughput
  - super-threaded
  - large-scale SIMD



# Stream computing

- A parallel processing model where a computational ***kernel*** is applied to a set of data (a ***stream***)
  - the kernel is applied to stream elements in parallel




- GPUs excel at this thanks to a large number of processing units and a parallel architecture

# Beyond stream computing

- Current GPUs offer functionality that goes beyond mere stream computing
- Shared memory and thread synchronization primitives eliminate the need for data independence
- Gather and scatter operations allow kernels to read and write data at arbitrary locations

# CUDA

- "Compute Unified Device Architecture
- A platform that exposes NVIDIA GPUs as general purpose *compute devices*
- Is CUDA considered GPGPU?
  - yes and no
    - CUDA can execute on devices with no graphics output capabilities (the NVIDIA Tesla product line) – these are not "GPUs", per se
    - however, if you are using CUDA to run some generic algorithms on your graphics card, you are indeed performing some ***G***eneral ***P***urpose computation on your ***G***raphics ***P***rocessing ***U***nit…

# What is CUDA used for?

- CUDA has been used in many different areas
  - options pricing in finance
  - electromagnetic simulations
  - fluid dynamics
  - GIS
  - geophysical data processing
  - 3D visualization solutions
  - …

- See http://www.nvidia.com/object/cuda_home_new.htm
  - long list of projects and speedups achieved

# Speedup

- What kind of speedup can I expect?
  - 0x – 2000x reported
  - 10x – considered typical (vs. multi-CPU machines)
  - >= 30x considered worthwhile

- Speedup depends on
  - problem structure
    - need many identical independent calculations
    - preferably sequential memory access
  - level of intimacy with hardware
  - time investment

# How to get running on the GPU?

- Easiest case: the package you are using already has a GPU-accelerated version. No programming needed.
- Medium case: your program spends most of its time in library routines which have GPU accelerated versions. Use libraries that take advantage of GPU acceleration. Small programming effort required.
- Hard case: You cannot take advantage of the easier two possibilities, so you must convert some of your code to CUDA or OpenCL
- Newly available OpenACC framework is an alternative that should make coding easier.

# GPU-enabled software

- A growing number of popular scientific software packages have now been accelerated for the GPU
- Using a GPU accelerated package requires no programming effort for the user
- Acceleration of Molecular Dynamics software has been particularly successful, with all major packages offering the GPU acceleration option

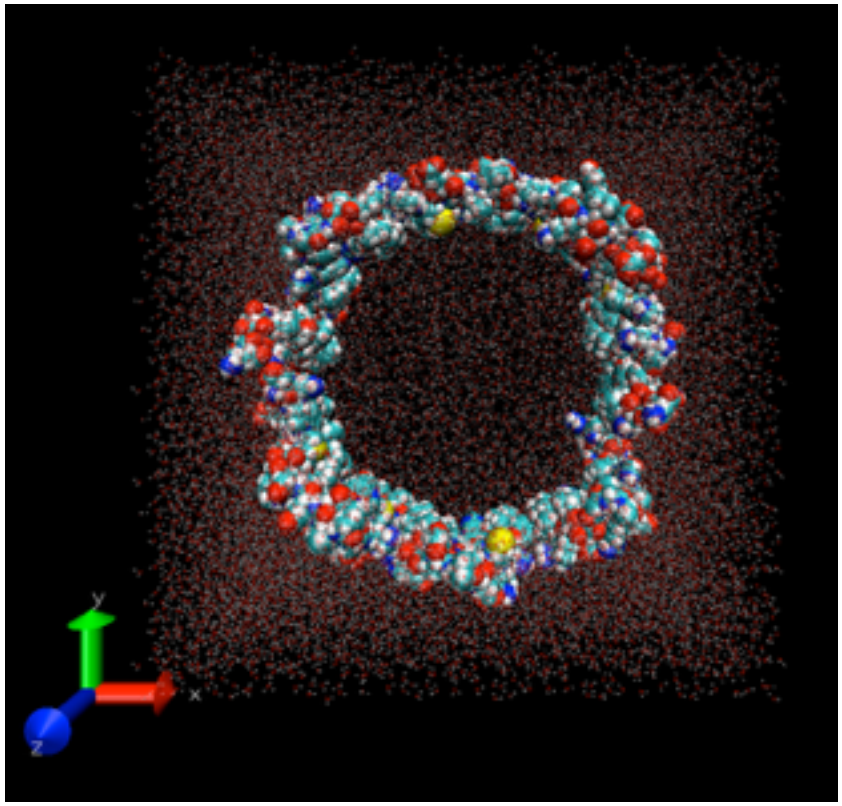
# NAMD

- http://www.ks.uiuc.edu/Research/namd/
- NAMD = Not (just)Another Molecular Dynamics program
- Free and open source
- Written using Charm++ parallel programming model
- Noted for its parallel efficiency

# NAMD performance on monk

- apoa1 standard NAMD benchmark, 92224 atoms simulated for 500 time steps, **wall time** in seconds:

| #threads | no GPU | 1 GPU | 2 GPU |
|---|---|---|---|
| 1 | 867.3 | 76.9 | 76.6 |
| 2 | 440.5 | 45.7 | 43.2 |
| 4 | 223 | 40.4 | 28.6 |
| 8 | 113.7 | 39.3 | 23.7 |



# NAMD performance on monk

- apoa1 standard NAMD benchmark, 92224 atoms simulated for 500 time steps, **speedup** over 1 thread/no GPU:

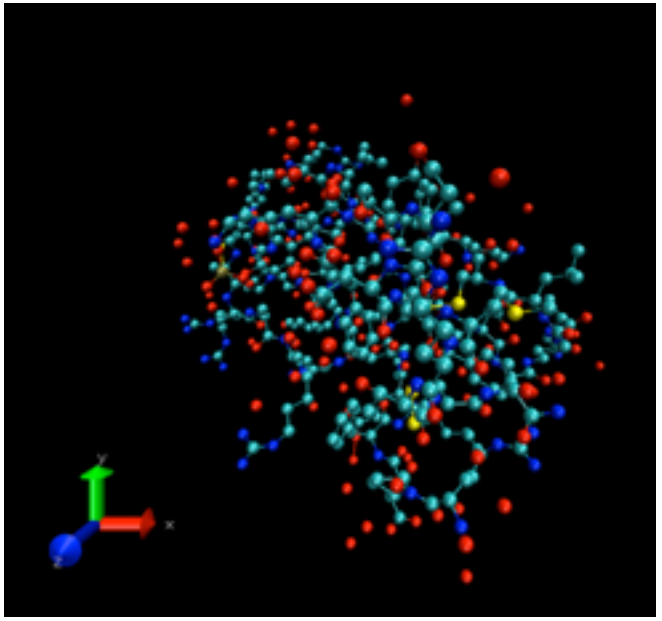
| #threads | no GPU | 1 GPU | 2 GPU |
|---|---|---|---|
| 1 | 1 | 11.3 | 11.3 |
| 2 | 2 | 19 | 20.1 |
| 4 | 3.9 | 21.5 | 30.3 |
| 8 | 7.7 | 22.1 | 36.6 |

- Speedup over 8-core/no GPU: 2.9 with 1 GPU, 4.8 with 2
- Most efficient: 2 runs of 4 core/1 GPU, speedup 2*21.5=43.0

# NAMD performance on monk

- bpti6 standard NAMD demo, 1101 atoms simulated for 21,000 time steps, **wall time** in seconds:

| #threads | no GPU | 1 GPU | 2 GPU |
|---|---|---|---|
| 1 | 202.6 | 48.9 | 48.7 |
| 2 | 107.4 | 33.8 | 31.7 |
| 4 | 56.2 | 31.7 | 28.4 |
| 8 | 30.8 | 34.6 | 29.1 |



- For smaller system GPU acceleration is less useful.
- Performance depends on system size!

# NAMD performance on monk

- bpti6 standard NAMD demo, 1101 atoms simulated for 21,000 time steps, **speedup** over 1 thread/no GPU:

| #threads | no GPU | 1 GPU | 2 GPU |
|---|---|---|---|
| 1 | 1 | 4.1 | 4.2 |
| 2 | 1.9 | 6 | 6.4 |
| 4 | 3.6 | 6.4 | 7.1 |
| 8 | 6.6 | 5.9 | 7 |

# Tesla M2070 ($1000+) vs GTX 570 ($300)

- apoa1 standard NAMD benchmark, 92224 atoms simulated for 500 time steps, wall time in seconds:

| #threads | no GPU | M2070 | GTX 570 |
|---|---|---|---|
| 1 | 867.3 | 76.9 | 73.2 |
| 2 | 440.5 | 45.7 | 38.4 |
| 4 | 223 | 40.4 | 33.3 |
| 8 | 113.7 | 39.3 | 32.8 |

# CUDA programming model

- The main CPU is referred to as the ***host***
- The compute device is viewed as a ***coprocessor*** capable of executing a large number of lightweight threads in parallel
- Computation on the device is performed by ***kernels***, functions executed in parallel on each data element
- Both the host and the device have their own ***memory***
  - the host and device cannot directly access each other's memory, but data can be transferred using the runtime API
- The host manages all memory allocations on the device, data transfers, and the invocation of kernels on the device

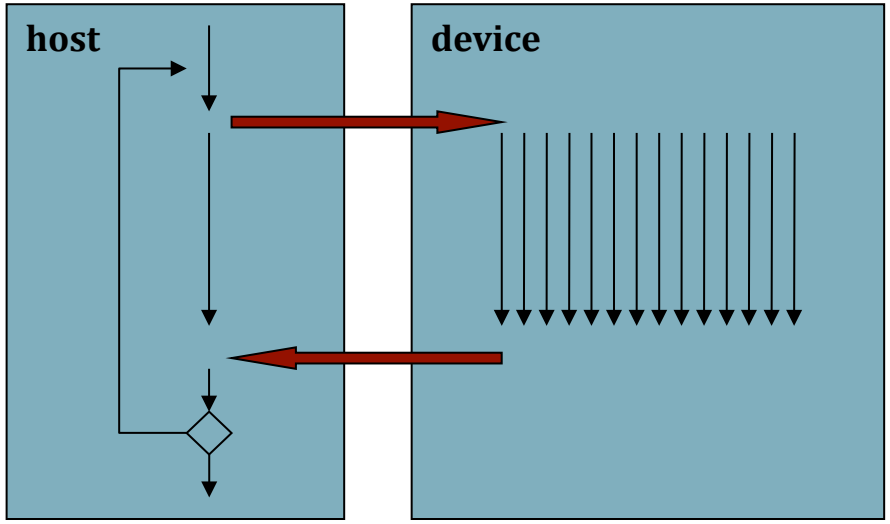
# GPU applications

- The GPU can be utilized in different capacities

- One is to use the GPU as a massively parallel coprocessor for number crunching applications
  - upload data and kernel to GPU
  - execute kernel
  - download results
  - CPU and GPU can execute asynchronously

- Some applications use the GPU for both data crunching and visualization
  - CUDA has bindings for OpenGL and Direct3D

# GPU as coprocessor

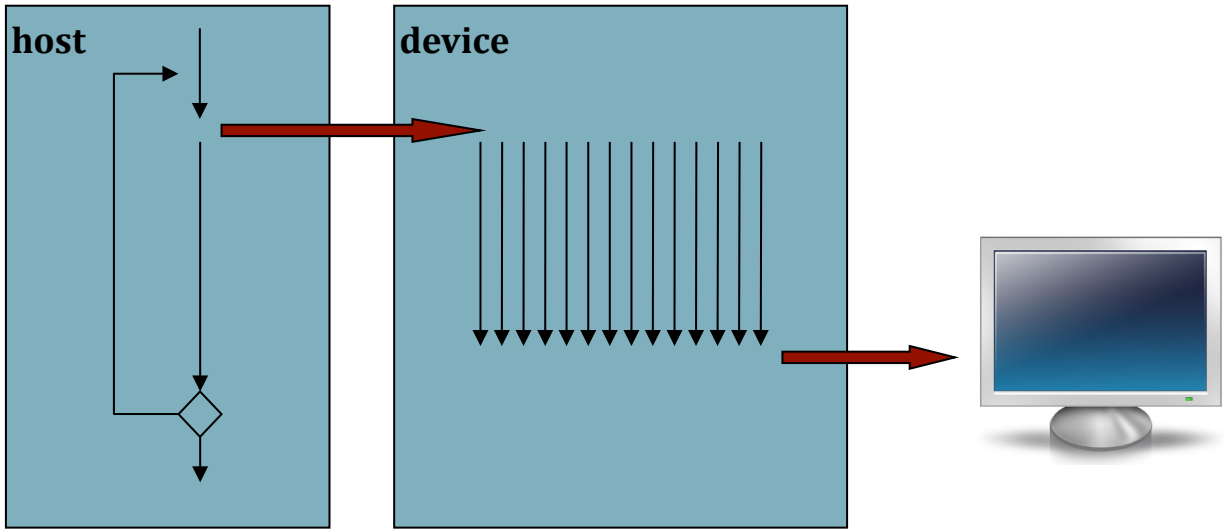



Kernel execution is asynchronous

Asynchronous memory transfers also available

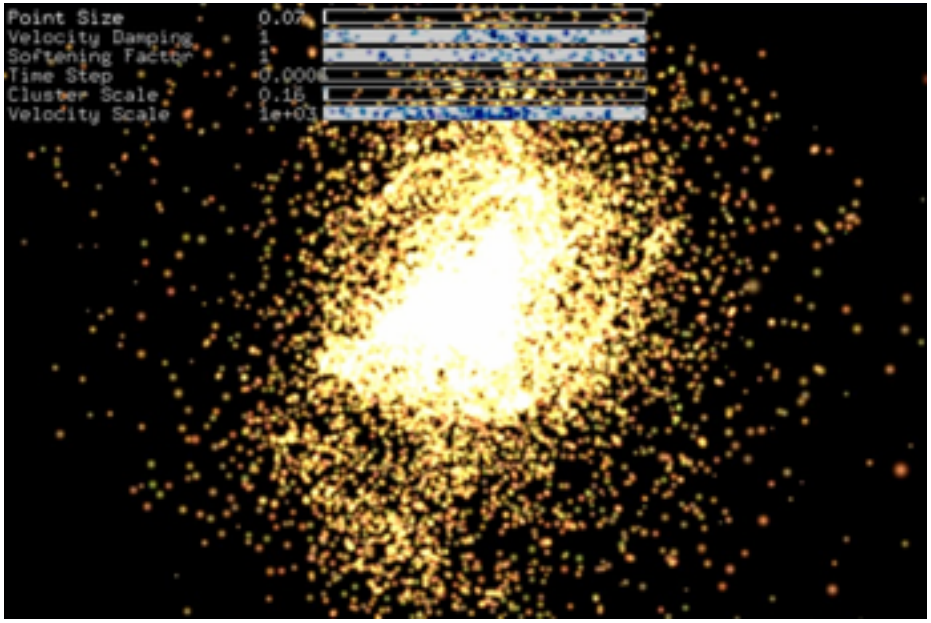
- Basic paradigm
  - host uploads inputs to device
  - host remains busy while device performs computation
    - prepare next batch of data, process previous results, etc.
  - host downloads results
- Can be iterative or multi-stage

# Simulation + visualization



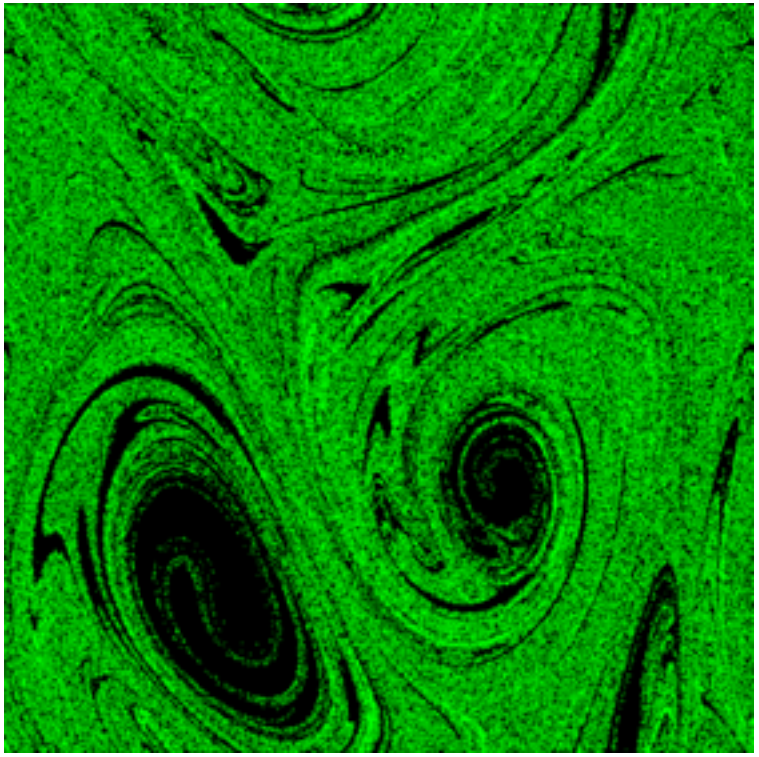

- Basic paradigm
  - host uploads inputs to device
  - host may remain busy while device performs computation
    - prepare next batch of data, etc.
  - results used on device for rendering, no download to host

# Simulation + visualization (cont.)




N-Body Demo

Fluids Demo



# SHARCNET GPU systems

- Always check our software page for latest info! See also:
  https://www.sharcnet.ca/help/index.php/GPU_Accelerated_Computing

- *angel.sharcnet.ca*

  11 NVIDIA Tesla S1070 GPU servers

  each with 4 GPUs + 16GB of global memory

  each GPU server connected to ***two*** compute nodes (2 4-core Xeon CPUs + 8GB RAM each)

  1 GPU per quad-core CPU; 1:1 memory ratio between GPUs/CPUs

- visualization workstations

  Some old and don't support CUDA, but some have up to date cards, check list at:

  https://www.sharcnet.ca/my/systems/index

# 2012 arrival - “monk” cluster

- 54 nodes, InfiniBand interconnect, 80 Tb storage
- Node:
  8 x CPU cores (Intel Xeon 2.26 GHz)
  48 GB memory
  2 x M2070 GPU cards
- Nvidia Tesla M2070 GPU
  “Fermi” architecture
  ECC memory protection
  L1 and L2 caches
  2.0 Compute Capability
  448 CUDA cores
  515 Gigaflops (DP)

# CUDA versions installed

- Different versions of CUDA available - choose one via modules

– on monk latest CUDA installed in `/opt/sharcnet/cuda/5.0.35/`

– sample projects in `/opt/sharcnet/cuda/5.0.35/samples`
  - copy to your work space (e.g. `/work/username/cuda_sdk`) & compile following instructions on the software page

    https://www.sharcnet.ca/help/index.php/CUDA

Development node: mon54 for interactive use, plus viz stations

# Output of device diagnostic program

```
   ...

   [ppomorsk@mon54:~/CUDA_day1/device_diagnostic] ./device_diagnostic.x
found 2 CUDA devices
   --- General Information for device 0 ---
Name:  Tesla M2070
Compute capability:  2.0
Clock rate:  1147000
Device copy overlap:  Enabled
Kernel execution timeout :  Disabled
   --- Memory Information for device 0 ---
Total global mem:  5636554752
Total constant Mem:  65536
Max mem pitch:  2147483647
Texture Alignment:  512
   --- MP Information for device 0 ---
Multiprocessor count:  14
Shared mem per mp:  49152
Registers per mp:  32768
Threads in warp:  32
Max threads per block:  1024
Max thread dimensions:  (1024, 1024, 64)
Max grid dimensions:  (65535, 65535, 65535)

   --- General Information for device 1 ---
Name:  Tesla M2070
...
```

# Submitting GPU jobs

- See GPU Accelerated Computing article on training wiki for maximum detail

– note: queue details (mpi vs. gpu – test queue oddities)

– To submit a job to gpu queue on angel

```
sqsub -q qpu --gpp=1 -n 1 -o out.txt -r 5m ./a.out
```

# Thread batching

- To take advantage of the multiple multiprocessors, kernels are executed as a ***grid*** of ***threaded blocks***

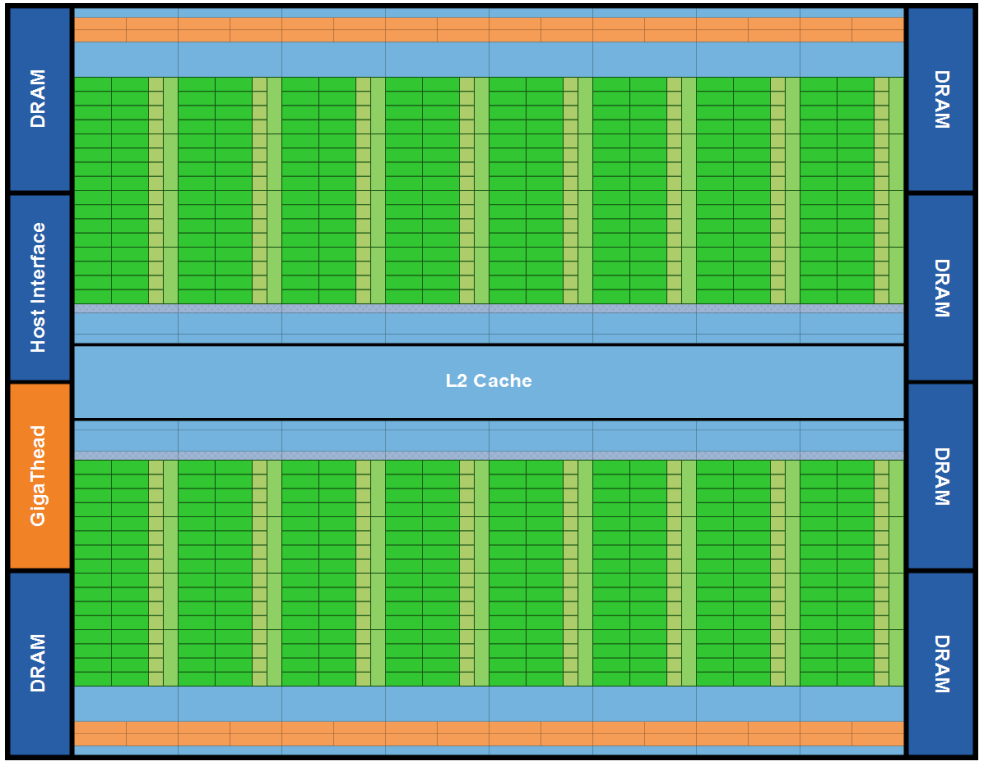
- All threads in a thread block are executed by a single multiprocessor

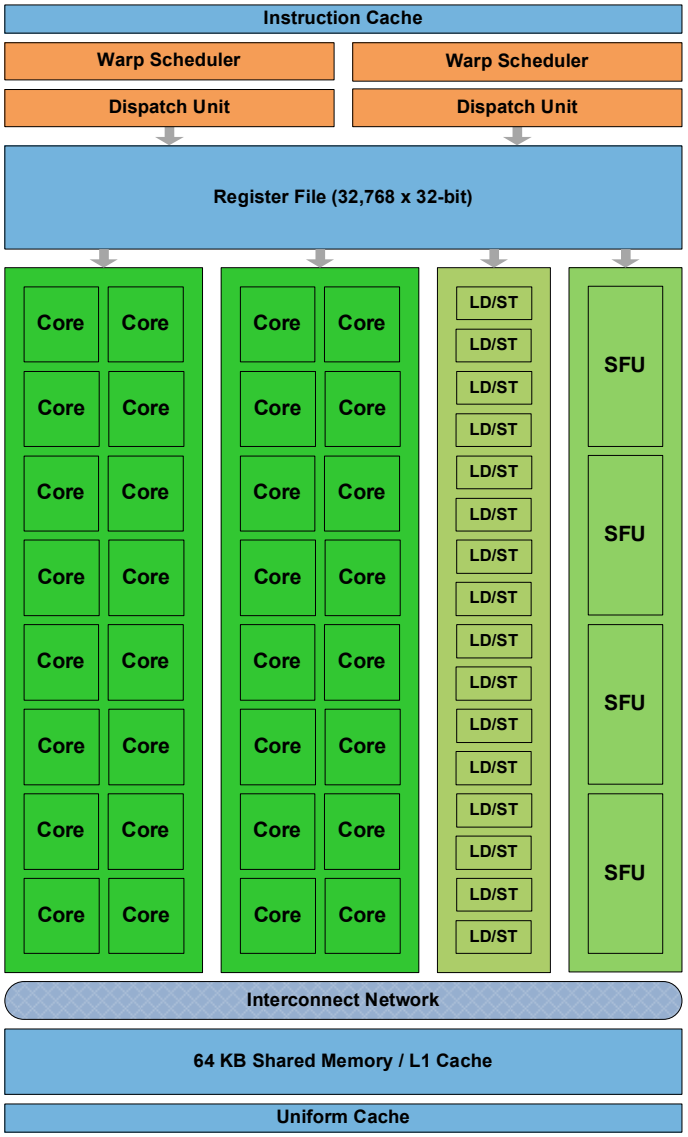
- The resources of a multiprocessor are divided among the threads in a block (registers, shared memory, etc.)
  - this has several important implications that will be discussed later

# Hardware basics

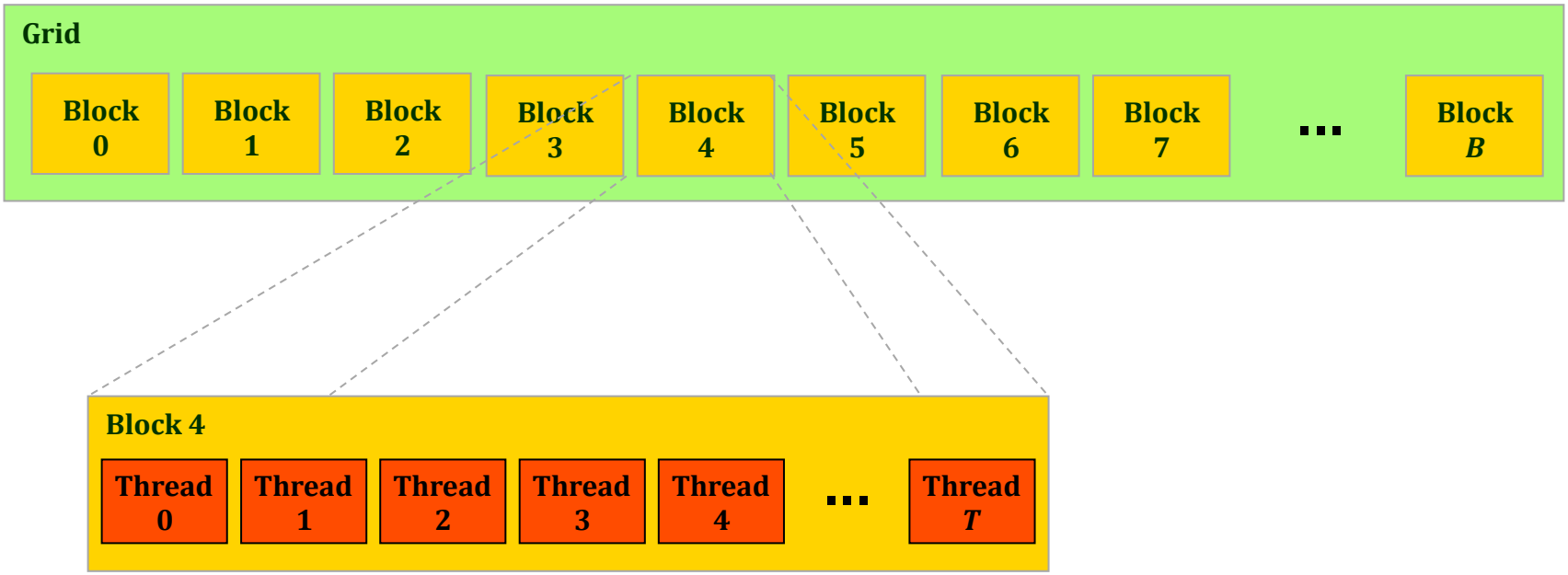
- The compute device is composed of a number of ***multiprocessors***, each of which contains a number of SIMD processors
  - Tesla M2070 has 14 multiprocessors (each with 32 CUDA cores)
- A multiprocessor can execute K ***threads*** in parallel physically, where K is called the ***warp size***
  - ***thread*** = instance of kernel
  - warp size on current hardware is 32 threads
- Each multiprocessor contains a large number of 32-bit ***registers*** which are divided among the active threads

# GPU Hardware architecture - NVIDIA Fermi

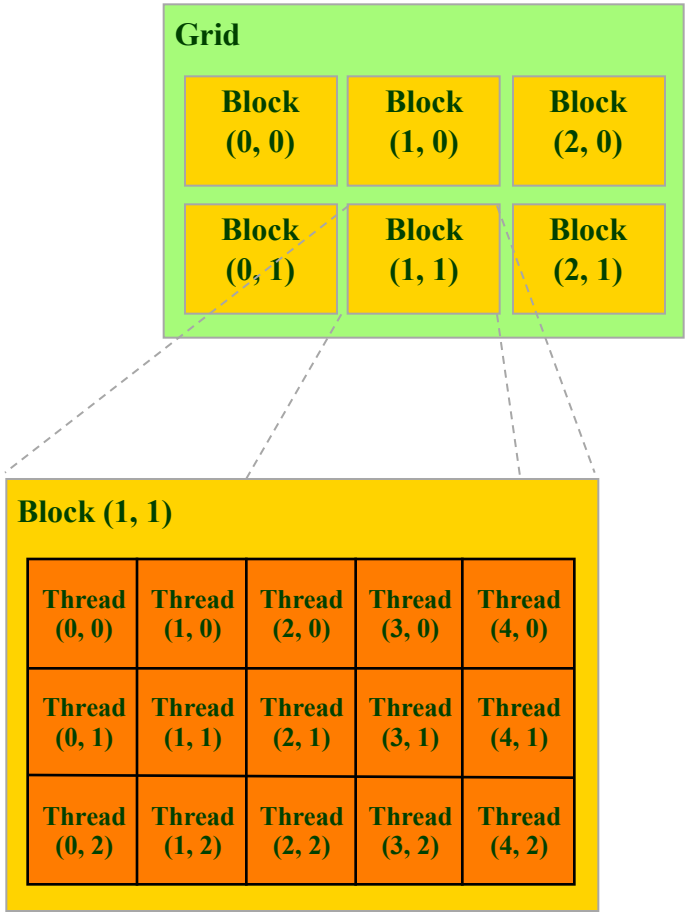


# Thread batching: 1D example



# Thread batching: 2D example



# Thread batching (cont.)

- At runtime, a thread can determine the block that it belongs to, the block dimensions, and the thread index within the block

- These values can be used to compute indices into input and output arrays

Introduction to GPU Programming: CUDA

# HELLO, CUDA!

# Language and compiler

- CUDA provides a set of extensions to the C programming language
  - new storage quantifiers, kernel invocation syntax, intrinsics, vector types, etc.
- CUDA source code saved in `.cu` files
  - host and device code and coexist in the same file
  - storage qualifiers determine type of code
- Compiled to object files using `nvcc` compiler
  - object files contain executable host and device code
- Can be linked with object files generated by other C/C++ compilers

# SAXPY

- SAXPY (Scalar Alpha X Plus Y) is a common linear algebra operation. It is a combination of scalar multiplication and vector addition:

$$y = \alpha \cdot x + y$$

  - $x$ and $y$ are vectors, $\alpha$ is a scalar
  - $x$ and $y$ can be arbitrarily large

# SAXPY: CPU version

- Here is SAXPY in vanilla C:

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

  - the CPU processes vector components sequentially using a for loop
  - note that `vecY` is an in-out parameter here

# SAXPY: CUDA version

- CUDA kernel function implementing SAXPY

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The **`__global__`** qualifier identifies this function as a kernel that executes on the device

# SAXPY: CUDA version (cont.)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- **`blockIdx`**, **`blockDim`** and **`threadIdx`** are built-in variables that uniquely identify a thread's position in the execution environment
  - they are used to compute an offset into the data array

# SAXPY: CUDA version (cont.)

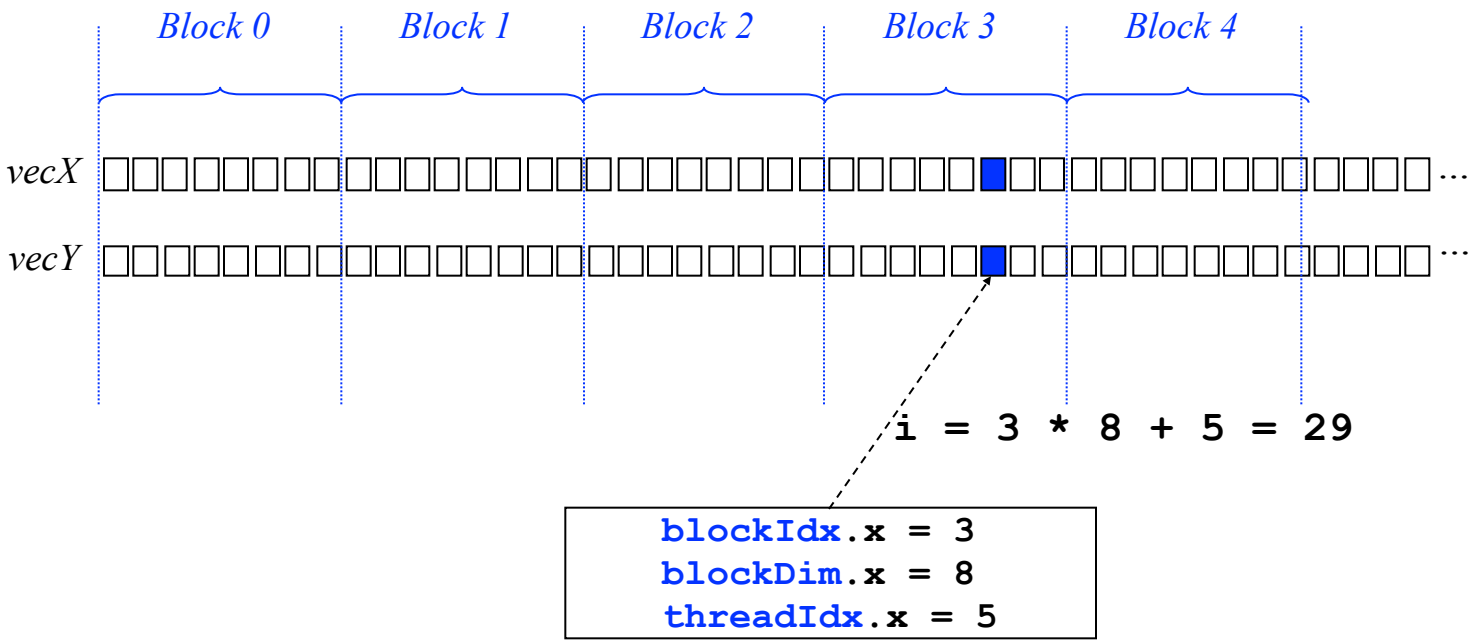
```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- The host specifies the number of blocks and block size during kernel invocation:

```
saxpy_gpu<<<numBlocks, blockSize>>>(y_d, x_d, alpha, n);
```

# Computing the index

```
i = blockIdx.x * blockDim.x + threadIdx.x;
if (i<n)
   vecY[i] = alpha * vecX[i] + vecY[i];
```



# Key differences

```
void saxpy_cpu(float *vecY, float *vecX, float alpha, int n)
{
     int i;

     for (i = 0; i < n; i++)
          vecY[i] = alpha * vecX[i] + vecY[i];
}
```

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- No need to explicitly loop over array elements – each element is processed in a separate thread
- The element index is computed based on block index, block width and thread index within the block

# Key differences

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   if (i<n)
      vecY[i] = alpha * vecX[i] + vecY[i];
}
```

- Could avoid testing whether $i < n$ if we knew $n$ is a multiple of block size (e.g. use padded arrays --- recall MPI_Scatter issues)

```
__global__ void saxpy_gpu(float *vecY, float *vecX, float alpha ,int n)
{
   int i;

   i = blockIdx.x * blockDim.x + threadIdx.x;
   vecY[i] = alpha * vecX[i] + vecY[i];
}
```

# Host code: overview

- The host performs the following operations:
  1. initialize device
  2. allocate and initialize input arrays in host DRAM
  3. allocate memory on device
  4. upload input data to device
  5. execute kernel on device
  6. download results
  7. check results
  8. clean-up

# Host code: initialization

```
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>

int main(int argc, char *argv[])
{
   float *x_host, *y_host;   /* arrays for computation on host*/
   float *x_dev, *y_dev;     /* arrays for computation on device */
   float *y_shadow;          /* host-side copy of device results */

   int n = 32*1024;
   float alpha = 0.5f;
   int nerror;

   size_t memsize;
   int i, blockSize, nBlocks;

/* here could add some code to check if GPU device is present */

...
```

# Host code: memory allocation

```
...

memsize = n * sizeof(float);

/* allocate arrays on host */

x_host = (float *)malloc(memsize);
y_host = (float *)malloc(memsize);
y_shadow = (float *)malloc(memsize);

/* allocate arrays on device */

cudaMalloc((void **) &x_dev, memsize);
cudaMalloc((void **) &y_dev, memsize);

/* add checks to catch any errors */

...
```

# Host code: upload data

```
...

   /* initialize arrays on host */

for ( i = 0; i < n; i++)
{
   x_host[i] = rand() / (float)RAND_MAX;
   y_host[i] = rand() / (float)RAND_MAX;
}

/* copy arrays to device memory (synchronous) */

cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice);
cudaMemcpy(y_dev, y_host, memsize, cudaMemcpyHostToDevice);

...
```

# Host code: kernel execution

```
...

   /* set up device execution configuration */
blockSize = 512;
nBlocks = n / blockSize + (n % blockSize > 0);

/* execute kernel (asynchronous!) */

saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

/* could add check if this succeeded */

/* execute host version (i.e. baseline reference results) */
saxpy_cpu(y_host, x_host, alpha, n);

...
```

# Host code: download results

```
...

/* retrieve results from device (synchronous) */
cudaMemcpy(y_shadow, y_dev, memsize, cudaMemcpyDeviceToHost);

/* ensure synchronization (cudaMemcpy is synchronous in most cases, but not all) */

cudaDeviceSynchronize();


/* check results */
nerror=0;
for(i=0; i < n; i++)
{
   if(y_shadow[i]!=y_host[i]) nerror=nerror+1;
}
printf("test comparison shows %d errors\n",nerror);

...
```

# Host code: clean-up

```
   ...

   /* free memory on device*/
   cudaFree(x_dev);
   cudaFree(y_dev);

  /* free memory on host */
   free(x_host);
   free(y_host);
   free(y_shadow);

   return 0;
} /* main */
```

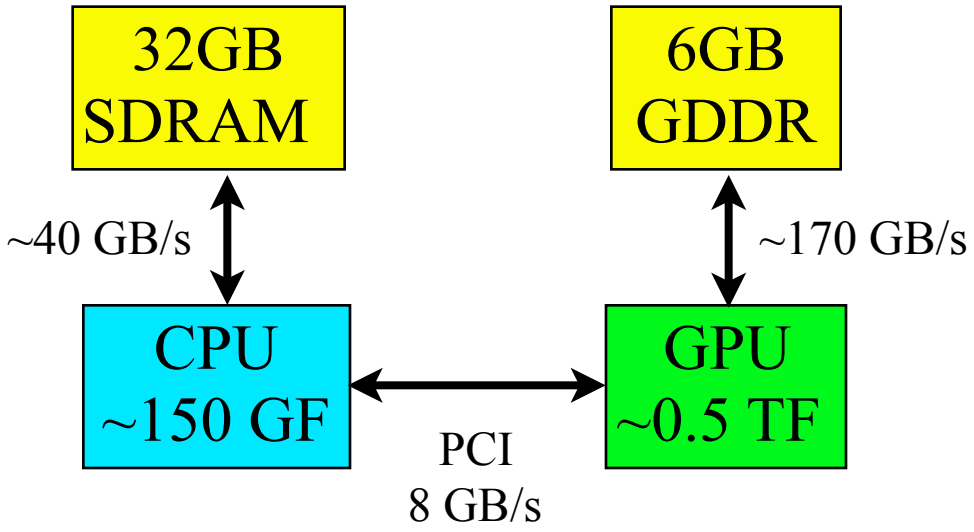
# Checking for errors in CUDA calls

```
 ...
 /* check CUDA API function call for possible error */
if (error = cudaMemcpy(x_dev, x_host, memsize, cudaMemcpyHostToDevice))
  {
    printf ("Error %d\n", error);
    exit (error);
  }

 ...

 saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);
 /* make sure kernel has completed*/
 cudaDeviceSynchronize();
 /* check for any error generated by kernel call*/
 if(error = cudaGetLastError())
  {
    printf ("Error detected after kernel %d\n", error);
    exit (error);
  }
```

# Compiling

- nvcc -arch=sm_20 -O2 program.cu -o program.x
- -arch=sm_20 means code is targeted at Compute Capability 2.0 architecture (what monk has)
- -O2 optimizes the CPU portion of the program
- There are no flags to optimize CUDA code
- Various fine tuning switches possible
- SHARCNET has a CUDA environment module preloaded. See what it does by executing: module show cuda
- add -lcublas to link with CUBLAS libraries

# Be aware of memory bandwidth bottlenecks




- The connection between CPU and GPU has low bandwidth
  - need to minimize data transfers
  - important to use asynchronous transfers if possible (overlap computation and transfer)

# Using pinned memory

- The transfer between host and device is very slow compared to access to memory within either the CPU or the GPU

- One way to speed it up by a factor of 2 or so is to use pinned memory on the host for memory allocation of array that will be transferred to the GPU

```
int main(int argc, char *argv[])
{
cudaMallocHost((void **) &a_host, memsize_input)
...
cudaFree(a_host);
```

# Timing GPU accelerated codes

- Presents specific difficulties because the CPU and GPU can be computing independently in parallel, i.e. asynchronously
- On the cpu can use standard function **gettimeofday(...)** (microsecond precision) and process the result
- If trying to time events on GPU with this function, must ensure synchronization
- This can be done with a call to cudaDeviceSynchronize()
- Memory copies to/from device are synchronized, so can be used for timing.
- Timing GPU kernels on the CPU may be insufficiently accurate

# Using mechanisms on the GPU for timing

- This is highly accurate on the GPU side, and very useful for optimizing kernels

```
...
   cudaEvent_t start, stop;
   float kernel_timer;
...
   cudaEventCreate(&start);
   cudaEventCreate(&stop);
   cudaEventRecord(start, 0);

   saxpy_gpu<<<nBlocks, blockSize>>>(y_dev, x_dev, alpha, n);

   cudaEventRecord(stop, 0);
   cudaEventSynchronize( stop );
   cudaEventElapsedTime( &kernel_timer, start, stop );

   printf("Test Kernel took %f ms\n",kernel_timer);
   cudaEventDestroy(start);
   cudaEventDestroy(stop);
```