Why would I use GPUs?

Pawel Pomorski, *HPC Software Analyst*
SHARCNET, University of Waterloo

[Email](ppomorsk@sharcnet.ca)
Overview

• Introduction to GPUs
• SHARCNET GPU hardware
• GPU enabled package: NAMD
• GPU accelerated library: CUBLAS
• Brief look at CUDA
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 OpenCL 1.0 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 look at 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
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

\[
\begin{align*}
\text{Input stream} & \quad 5 \quad 1 \quad 3 \quad 8 \quad 2 \quad 3 \quad 6 \quad 7 \quad 7 \quad 3 \quad 4 \quad 5 \\
\text{Kernel} & \quad y_i = x_i + 1 \\
\text{Output stream} & \quad 6 \quad 2 \quad 4 \quad 9 \quad 3 \quad 4 \quad 7 \quad 8 \quad 8 \quad 4 \quad 5 \quad 6
\end{align*}
\]

- 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 far 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
Speedup

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

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

- **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**

Kernel execution is asynchronous

Asynchronous memory transfers also available
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
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)
SHARCNET GPU systems

• Always check our software page for latest info! See also:

• 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

  – 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
New 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)
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 or monk

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

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

<table>
<thead>
<tr>
<th>#threads</th>
<th>no GPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>867.3</td>
<td>76.9</td>
<td>76.6</td>
</tr>
<tr>
<td>2</td>
<td>440.5</td>
<td>45.7</td>
<td>43.2</td>
</tr>
<tr>
<td>4</td>
<td>223.0</td>
<td>40.4</td>
<td>28.6</td>
</tr>
<tr>
<td>8</td>
<td>113.7</td>
<td>39.3</td>
<td>23.7</td>
</tr>
</tbody>
</table>
NAMD performance on monk

- apoA1 standard NAMD benchmark, 92224 atoms simulated for 500 time steps, **speedup** over 1 thread/no GPU:

<table>
<thead>
<tr>
<th>#threads</th>
<th>no GPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>11.3</td>
<td>11.3</td>
</tr>
<tr>
<td>2</td>
<td>2.0</td>
<td>19.0</td>
<td>20.1</td>
</tr>
<tr>
<td>4</td>
<td>3.9</td>
<td>21.5</td>
<td>30.3</td>
</tr>
<tr>
<td>8</td>
<td>7.7</td>
<td>22.1</td>
<td>36.6</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>#threads</th>
<th>no GPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>202.6</td>
<td>48.9</td>
<td>48.7</td>
</tr>
<tr>
<td>2</td>
<td>107.4</td>
<td>33.8</td>
<td>31.7</td>
</tr>
<tr>
<td>4</td>
<td>56.2</td>
<td>31.7</td>
<td>28.4</td>
</tr>
<tr>
<td>8</td>
<td>30.8</td>
<td>34.6</td>
<td>29.1</td>
</tr>
</tbody>
</table>

- 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:

<table>
<thead>
<tr>
<th>#threads</th>
<th>no GPU</th>
<th>1 GPU</th>
<th>2 GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>4.1</td>
<td>4.2</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>6.0</td>
<td>6.4</td>
</tr>
<tr>
<td>4</td>
<td>3.6</td>
<td>6.4</td>
<td>7.1</td>
</tr>
<tr>
<td>8</td>
<td>6.6</td>
<td>5.9</td>
<td>7.0</td>
</tr>
</tbody>
</table>
Tesla M2070 ($1000+) vs GTX 570 ($300)

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

<table>
<thead>
<tr>
<th>#threads</th>
<th>no GPU</th>
<th>M2070</th>
<th>GTX 570</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>867.3</td>
<td>76.9</td>
<td>73.2</td>
</tr>
<tr>
<td>2</td>
<td>440.5</td>
<td>45.7</td>
<td>38.4</td>
</tr>
<tr>
<td>4</td>
<td>223.0</td>
<td>40.4</td>
<td>33.3</td>
</tr>
<tr>
<td>8</td>
<td>113.7</td>
<td>39.3</td>
<td>32.8</td>
</tr>
</tbody>
</table>
GPU-accelerated libraries

- A number of these are available to address common scientific problems
- Linear algebra libraries
- FFTW libraries
- Parallel algorithms and data structure libraries
- Image processing primitives
- Random number generation
Linear algebra on the GPU

• Linear algebra on the CPU: BLAS, LAPACK
• GPU analogues: CUBLAS, CULA
•CUSPARSE library for sparse matrices
• Use of highly optimised libraries is always better than writing your own code, especially since GPU codes cannot yet be efficiently optimized by compilers to achieve acceptable performance
• Writing efficient GPU code requires special care and understanding the peculiarities of underlying hardware
CUBLAS

- Implementation of BLAS (Basic Linear Algebra Subprograms) on top of CUDA
- Included with CUDA (hence free)
- Workflow:
  1. allocate vectors and matrices in GPU memory
  2. fill them with data
  3. call sequence of CUBLAS functions
  4. transfer results from GPU memory to host
- Helper functions to transfer data to/from GPU provided
Error checks

• in following example most error checks were removed for clarity
• each CUBLAS function returns a status object containing information about possible errors
• It’s very important these objects to catch errors, via calls like this:

```c
if (status != CUBLAS_STATUS_SUCCESS) {
    print diagnostic information and exit}
```
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
Initialize program

```c
#include <cuda.h> /* CUDA runtime API */
#include <cstdio>
#include <cublas_v2.h>

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 */

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

    size_t memsize;
    int i;

    /* could add device detection here */

    memsize = n * sizeof(float);
}
Allocate memory on host and device

```c
/* allocate arrays on host */

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

/* allocate arrays on device */

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

/* 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);
```
Call CUBLAS function

cublasHandle_t handle;
cublasStatus_t status;

status = cublasCreate(&handle);

int stride = 1;
status = cublasSaxpy(handle, n, &alpha, x_dev, stride, y_dev, stride);

if (status != CUBLAS_STATUS_SUCCESS)
{
    printf("Error in CUBLAS routine \n");
    exit(20);
}

status = cublasDestroy(handle);
Retrieve computed data and finish

```c
/* retrieve results from device (synchronous) */
cudaMemcpy(y_host, y_dev, memsize, cudaMemcpyDeviceToHost);

/* use data in y_host*/

/* free memory */
cudaFree(x_dev);
cudaFree(y_dev);
free(x_host);
free(y_host);

return 0;
```
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
Performance of CUBLAS on monk

- test case: multiplying two single precision matrices: (16384 x 32) and (32 x 16384)
- Using MKL CBLAS_SGEMM routine for CPU calculation and CUBLAS cublasSgemm routine for GPU calculation

<table>
<thead>
<tr>
<th></th>
<th>time (s)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 CPU (serial)</td>
<td>1.310</td>
<td>1.0</td>
</tr>
<tr>
<td>8 CPU (threaded)</td>
<td>0.309</td>
<td>4.2</td>
</tr>
<tr>
<td>1 GPU (with copy)</td>
<td>0.225</td>
<td>5.8</td>
</tr>
<tr>
<td>1 GPU (without copy)</td>
<td>0.049</td>
<td>26.7</td>
</tr>
</tbody>
</table>

- GPU is 6.3 times faster than multicore CPU (if transfers not considered)
Rough performance estimate

- Common libraries typically offer 5 to 10 fold speedup on the GPU vs a multicore processor
- System size must be sufficiently large to achieve this speedup
- Memory transfer costs are not included. This means programmer must keep data on the GPU and apply multiple library operations to it
A very quick look at CUDA

- Consider attending our Summer School to learn more
  - if you can’t wait, plenty of materials online available
  - good book to start: *CUDA by example: an introduction to general-purpose GPU programming*
Thread batching: 1D example
Thread batching: 2D example

<table>
<thead>
<tr>
<th>Block (0, 0)</th>
<th>Block (1, 0)</th>
<th>Block (2, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block (0, 1)</td>
<td>Block (1, 1)</td>
<td>Block (2, 1)</td>
</tr>
</tbody>
</table>

Block (1, 1)

<table>
<thead>
<tr>
<th>Thread (0, 0)</th>
<th>Thread (1, 0)</th>
<th>Thread (2, 0)</th>
<th>Thread (3, 0)</th>
<th>Thread (4, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread (0, 1)</td>
<td>Thread (1, 1)</td>
<td>Thread (2, 1)</td>
<td>Thread (3, 1)</td>
<td>Thread (4, 1)</td>
</tr>
<tr>
<td>Thread (0, 2)</td>
<td>Thread (1, 2)</td>
<td>Thread (2, 2)</td>
<td>Thread (3, 2)</td>
<td>Thread (4, 2)</td>
</tr>
</tbody>
</table>
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
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:

```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

```c
__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