High-Performance Computing Using GPUs

Luca Caucci
caucci@email.arizona.edu

Center for Gamma-Ray Imaging

November 7, 2012

Outline

- Why GPUs? What is CUDA?
- The CUDA programming model
- Anatomy of a CUDA program
- An application in medical imaging
- Conclusions

Why GPUs? What is CUDA?

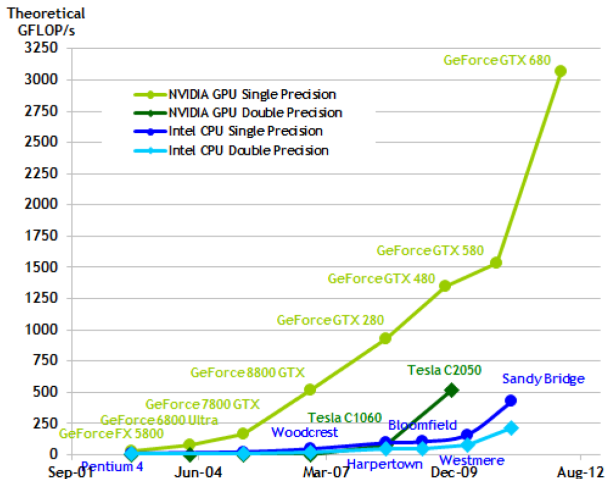
- GPU stands for "graphics processing unit"
- A GPU is a specialized computing device that offloads and accelerates graphics rendering from the CPU
- GPUs are very efficient at manipulating computer graphics and they are highly parallel (hundreds of cores)
- Originally designed for the entertainment industry (video coding, 3D rendering, games, . . .), they have become suited for general-purpose complex algorithms as well

Why GPUs? What is CUDA?

GPU Card	Number Cores	Total Memory	Single Prec. GFLOP/s	Price
GeForce GTX 480	480	1.5 GB	1344	$200
GeForce GTX 580	512	3 GB	1581	$530
GeForce GTX 690	3072	4 GB	5622	$1000
Tesla C2075	448	6 GB	1288	$2000
Tesla K10	3072	8 GB	5340	$3400
Tesla K20	— *To Be Announced* —			

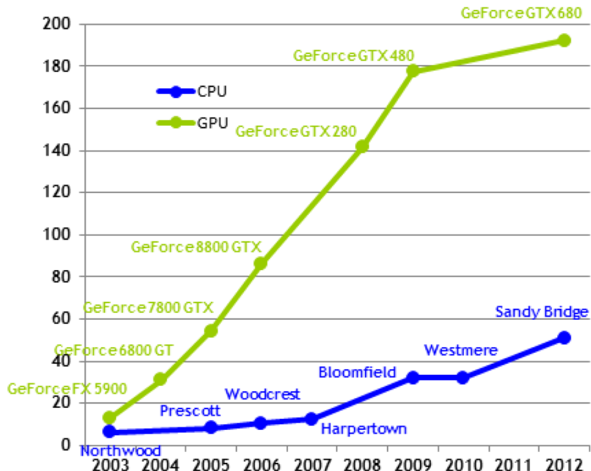
Why GPUs? What is CUDA?

Why GPUs? What is CUDA?



http://www.nvidia.com/

Why GPUs? What is CUDA?



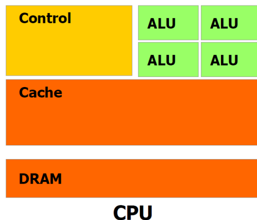
http://www.nvidia.com/

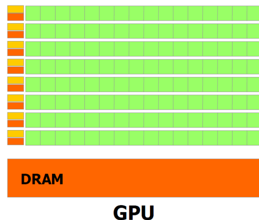
Why GPUs? What is CUDA?

- To satisfy the need of a convenient way to develop code for execution on a GPU device, NVIDIA developed CUDA
- CUDA stands for "compute unified device architecture"
- CUDA gives developers access to the instruction set and memory of the parallel computational elements of GPUs
- CUDA is a minimal extension to C/C++
- Many threads run in parallel slowly (rather than executing a single thread very fast, as on a CPU)
- CUDA SDK includes CUFFT, CUBLAS libraries, sparse matrices, random numbers, . . .

Why GPUs? What is CUDA?

- GPU:
  - Threads are extremely lightweight
  - Very little creation/scheduling overhead
  - Threads scheduled by the hardware
  - 100's or 1000's threads for full efficiency
- CPU:
  - Usually, OS schedules threads
  - Multi-core CPUs need only a few threads



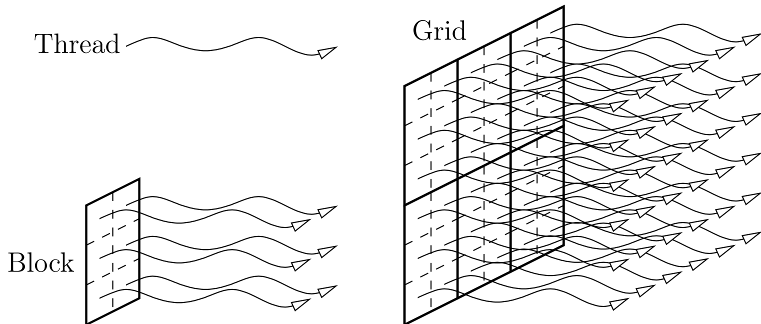


http://www.nvidia.com/

The CUDA programming model

- Data-parallel portions of an application are executed as *kernels*, which run in parallel on many *threads*
- Threads are organized in a hierarchy of *grids* of thread *blocks*
- Blocks can have up to 3 dimensions and contain up to 1024 threads. Threads in the same block can share data via shared memory
- Grids can have up to 2 dimensions and 65535×65535 blocks. No communication between blocks

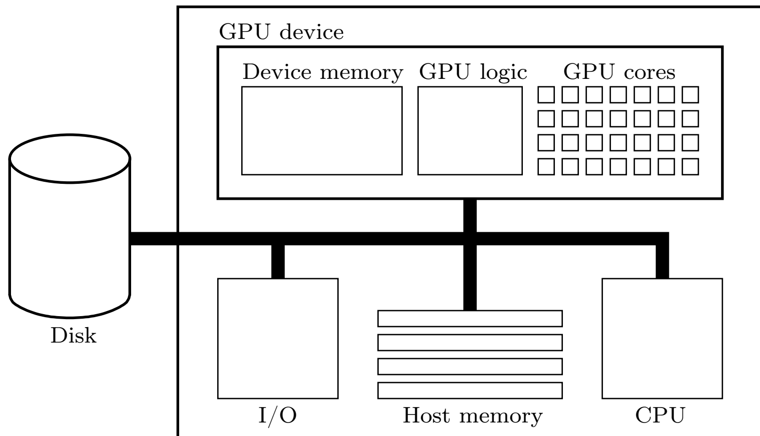
The CUDA programming model



The CUDA programming model

- In CUDA, threads execute on a physically separated *device*, that operates as a coprocessor to the *host*
- Both the host and the device have separate memory spaces, called *host memory* and *device memory*
- Device memory includes *shared*, *global*, *constant*, *texture*, . . . memories
- A thread can only access device memory
- Calls to the *CUDA runtime* (a library) allow manage device memory, and data transfer between host and device memory

The CUDA programming model



The CUDA programming model

- Value type qualifiers:
  - `__device__`: declares a variable that resides in the device memory
  - `__host__`: declares a variable that resides in the host memory (default)
  - `__shared__`: declares a variable that resides in the shared memory space of a thread block
  - `__constant__`: declares a variable that resides in constant memory space on the device
- Function type qualifiers:
  - `__global__`: runs on device, called from host code
  - `__device__`: runs on device, called from device code
  - `__host__`: runs on host, called from host code (default)

The CUDA programming model

Memory	Location	Cached?	Access	Scope	Lifetime
Local	Off-chip	No	R/W	One thread	Thread
Shared	On-chip	N/A	R/W	All threads in block	Block
Global	Off-chip	No	R/W	All threads and host	Application
Constant	Off-chip	Yes	R	All threads and host	Application
Texture	Off-chip	Yes	R	All threads and host	Application

- Use shared memory to improve performance (trade-off: 16 or 48 KB of shared memory per block)
- Global memory very slow; recalculation instead of retrieve

The CUDA programming model

- Built-in vector types: `char`n, `uchar`n, `short`n, `ushor`n, `int`n, `uint`n, `long`n, `ulong`n, `float`n, `double1`, `double2`, for $n = 1, \ldots, 4$
- These are 1D, 2D, 3D, 4D vector types. They are structures with `.x`, `.y`, `.z`, `.w` fields
- Built-in variables: `gridDim`, `blockIdx`, `blockDim`, `threadIdx`. They are of type `dim3`, which is the same as `uint3`
- Built-in variables used by threads to calculate indexes in arrays, matrices, etc.

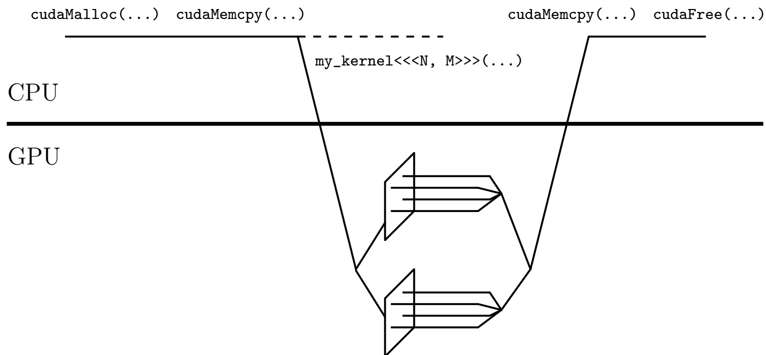
The CUDA programming model

- `<<<...>>>` is used to invoke a kernel from the host code
- The `<<<...>>>` CUDA syntax instructs the hardware to generate and run threads on the device
- For example, `my_kernel<<<N, M>>>(...)`
- `N`, of type `dim3` or `int`, tells the grid size
- `M`, of type `dim3` or `int`, tells the block size
- Thread scheduling is performed automatically by the hardware

Anatomy of a CUDA program

- General idea:
  - Execute serial code (such load data from disk) on host
  - Allocate memory on device
  - Copy data from host memory to device memory
  - Call kernel using `<<<...>>>` syntax
  - Copy result from device memory to host memory
  - Release memory allocated on device
- Include `cuda.h` for the CUDA runtime
- Compile using the NVIDIA `nvcc` compiler

Anatomy of a CUDA program



Anatomy of a CUDA program

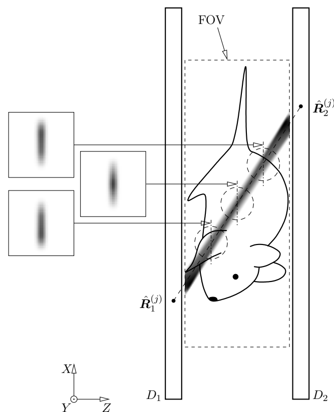
- An example of a CUDA application: add two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ together to produce $\boldsymbol{c}$
- The kernel code is shown below

```
__global__ void add_kernel(float *a, float *b, float *c, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < n) {
    c[i] = a[i] + b[i];
  }
  return;
}
```

Anatomy of a CUDA program


```
#include <cuda.h>

__global__ void add_kernel(float *a, float *b, float *c, int n);

int main(int argv, char **argv) {
  float a[1024], b[1024], c[1024];
  float *a_dev, *b_dev, *c_dev;
  int i;

  ...  // Fill out arrays a and b

  cudaMalloc((void **) (& a_dev), 1024 * sizeof(float));
  cudaMalloc((void **) (& b_dev), 1024 * sizeof(float));
  cudaMalloc((void **) (& c_dev), 1024 * sizeof(float));
  cudaMemcpy(a_dev, a, 1024 * sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(b_dev, b, 1024 * sizeof(float), cudaMemcpyHostToDevice);
  add_kernel<<<4, 256>>>(a_dev, b_dev, c_dev, 1024);
  cudaMemcpy(c, c_dev, 1024 * sizeof(float), cudaMemcpyDeviceToHost);
  cudaFree(a_dev); cudaFree(b_dev); cudaFree(c_dev);

  ...  // Use data in array c

  return(0);
}
```

An application in medical imaging



- Simple PET setup
- Gamma-ray photons emitted by object
- Points $\hat{\boldsymbol{R}}_1^{(j)}$ and $\hat{\boldsymbol{R}}_2^{(j)}$ estimated from PMT data
- Define $\hat{\boldsymbol{A}}^{(j)} = \left(\hat{\boldsymbol{R}}_1^{(j)}, \hat{\boldsymbol{R}}_2^{(j)}\right)$
- Build list $\hat{\mathcal{A}} = \left\{\hat{\boldsymbol{A}}^{(1)}, \ldots, \hat{\boldsymbol{A}}^{(J)}\right\}$

- We want to reconstruct object $\boldsymbol{f}$ from list $\hat{\mathcal{A}}$
- Use the list-mode (LM) MLEM algorithm:

$$\hat{f}_n^{(k+1)} = \hat{f}_n^{(k)} \left\{ \frac{1}{\tau} \sum_{j=1}^{J} \frac{\mathrm{pr}(\hat{\boldsymbol{A}}^{(j)} \mid n)}{\sum_{n'=1}^{N} \mathrm{pr}(\hat{\boldsymbol{A}}^{(j)} \mid n') s_{n'} \hat{f}_{n'}^{(k)}} \right\}$$

- Need to calculate probability of $\hat{\boldsymbol{A}}^{(j)}$ given emission within n^{th} voxel:

$$\mathrm{pr}(\hat{\boldsymbol{A}}^{(j)} \mid n) = \frac{\mu_{\mathrm{pe}}^2}{4\pi Z_{\max}^2} \times$$

$$\times \int_{D_1} \mathrm{pr}(\hat{\boldsymbol{R}}_1^{(j)} \mid \boldsymbol{R}_1^{(j)}) \frac{e^{-\mu_{\mathrm{tot}} \Delta_1(\boldsymbol{R}_1^{(j)}; \boldsymbol{r}_n)}}{|\boldsymbol{R}_1^{(j)} - \boldsymbol{r}_n|^2} \int_{D_2} \mathrm{pr}(\hat{\boldsymbol{R}}_2^{(j)} \mid \boldsymbol{R}_2^{(j)}) e^{-\mu_{\mathrm{tot}} \Delta_2(\boldsymbol{R}_2^{(j)}; \boldsymbol{r}_n)} \times$$

$$\times \int_{-\infty}^{\infty} \psi_{D_2}\big(\boldsymbol{R}_1^{(j)} + (\boldsymbol{r}_n - \boldsymbol{R}_1^{(j)})\ell\big) \delta_{\mathrm{Dir}}\big(\boldsymbol{R}_2^{(j)} - \boldsymbol{R}_1^{(j)} - (\boldsymbol{r}_n - \boldsymbol{R}_1^{(j)})\ell\big) \,\mathrm{d}\ell \,\mathrm{d}^3 \boldsymbol{R}_2^{(j)} \,\mathrm{d}^3 \boldsymbol{R}_1^{(j)}$$

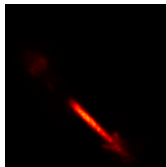
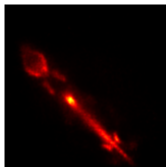
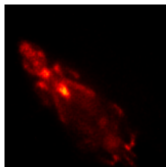
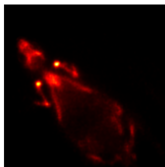
- Integrals amenable to GPU computation

An application in medical imaging

Computing Platform	Task: $\mathrm{pr}(\hat{A}^{(j)} \mid n)$	Task: LMMLEM
Intel® Xeon® L5506 2.13 GHz	9347.17 s 10.70 events/s	2.17 s 0.22 s/iter
NVIDIA Tesla C2050, 2 devices	30.30 s (308.49×) 3311.45 events/s	*n/a*
NVIDIA Tesla C2050, 4 devices	15.52 s (602.27×) 6442.47 events/s	*n/a*
NVIDIA Tesla C2050, 6 devices	11.22 s (833.08×) 8911.21 events/s	*n/a*
NVIDIA Tesla C2050, 8 devices	9.74 s (959.67×) 10264.06 events/s	*n/a*

An application in medical imaging

- ^{18}F-NaF bone scan for a normal mouse
- $J = 2280715$ elements in list $\hat{\mathcal{A}}$
- $88\ \mathrm{mm} \times 88\ \mathrm{mm} \times 32\ \mathrm{mm}$ FOV size
- $550\ \mu\mathrm{m} \times 550\ \mu\mathrm{m} \times 500\ \mu\mathrm{m}$ voxel size

Conclusions

- GPU hardware viable for high-performance computing (many $\times 100$'s speedup)
- Many products on the market
- Prices constantly falling; performance constantly increasing
- CUDA: minimal extensions to C/C++
- CUDA programming model is easy and scales well
- Tools to use GPUs with Matlab, *Mathematica*, ...

Conclusions

- Enormous potential for number crunching applications:
  - SPECT, PET, CT, MRI, ...
  - Physics, chemistry, biology, material science, ...
  - Matrix operations
  - Video/audio manipulation
  - ...
- Lots of resources:
  - `www.nvidia.com`
  - NVIDIA CUDA™ C Programming Guide
  - Examples
  - Books
  - Forums
  - ...

Questions?