Statistical Computation using GPU's

Paul Baines

Department of Statistics
University of California, Davis

May 1st, 2012

Statistics & GPU's

Overview for today:

- What is a GPU?
- How is it different from a CPU?
- How to use GPU's for scientific computation
- When to use GPU's for scientific computation

Credit: Lots of slides taken from the web!

Background and History

GPU's (graphical processing units) are specialized units designed for rendering computer graphics.

They work very differently from CPU's (central processing units) which perform the bulk of the tasks on your computer.

Background and History

GPU's (graphical processing units) are specialized units designed for rendering computer graphics.

They work very differently from CPU's (central processing units) which perform the bulk of the tasks on your computer.

Rendering high-definition computer graphics quickly and smoothly requires billions of simple calculations to be performed in seconds. GPU's are designed specifically for this task.

Background and History

GPU's (graphical processing units) are specialized units designed for rendering computer graphics.

They work very differently from CPU's (central processing units) which perform the bulk of the tasks on your computer.

Rendering high-definition computer graphics quickly and smoothly requires billions of simple calculations to be performed in seconds. GPU's are designed specifically for this task.

In recent years, there has been a great deal of progress in using GPU's for more general purpose calculations, not just graphics.

NVIDIA (and their language CUDA) are at the forefront of this effort.

Background and History

GPU's (graphical processing units) are specialized units designed for rendering computer graphics.

They work very differently from CPU's (central processing units) which perform the bulk of the tasks on your computer.

Rendering high-definition computer graphics quickly and smoothly requires billions of simple calculations to be performed in seconds. GPU's are designed specifically for this task.

In recent years, there has been a great deal of progress in using GPU's for more general purpose calculations, not just graphics.

NVIDIA (and their language CUDA) are at the forefront of this effort.

Before we talk specifics... what you need to know...

Types of Parallelism

Two main types of parallelism:

- Type I: Task Parallelism: Idea is to parallelize different tasks that do not depend on other uncompleted tasks.
  The taks being parallelized can be completely different.

  Example: Computing multivariate normal densities:

Types of Parallelism

Two main types of parallelism:

- Type I: Task Parallelism: Idea is to parallelize different tasks that do not depend on other uncompleted tasks.
  The taks being parallelized can be completely different.

  Example: Computing multivariate normal densities:
  (1) Compute Cholesky decomposition

Types of Parallelism

Two main types of parallelism:

- Type I: Task Parallelism: Idea is to parallelize different tasks that do not depend on other uncompleted tasks.
  The taks being parallelized can be completely different.

  Example: Computing multivariate normal densities:

  (1) Compute Cholesky decomposition
  (2a) Compute inverse of Cholesky factor
  (2b) Compute determinant of Cholesky factor

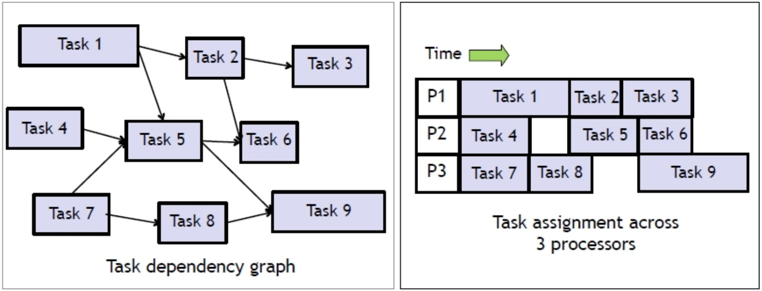
Types of Parallelism

Two main types of parallelism:

- Type I: Task Parallelism: Idea is to parallelize different tasks that do not depend on other uncompleted tasks.
  The taks being parallelized can be completely different.

  Example: Computing multivariate normal densities:

  (1) Compute Cholesky decomposition
  (2a) Compute inverse of Cholesky factor
  (2b) Compute determinant of Cholesky factor
  (3) Finish computing the density



Credit: CS264 (N. Pinto)

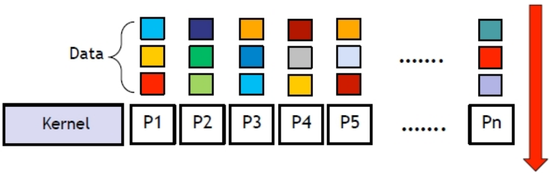
Parallelism II: Data Parallelism

GPU's are not useful for task parallelism, for are useful for a different kind of parallelism: data parallelism.

Type II: Data Parallelism:
Perform the same task on multiple pieces of data.

Examples:

- Matrix multiplication: same task (multiplication), on multiple pieces of data (matrix elements)
- Numerical integration: same task (function evaluation), on multiple pieces of data (integration grid)



Credit: CS264 (N. Pinto)

CPU vs. GPU

Credit: CS264 (N. Pinto)

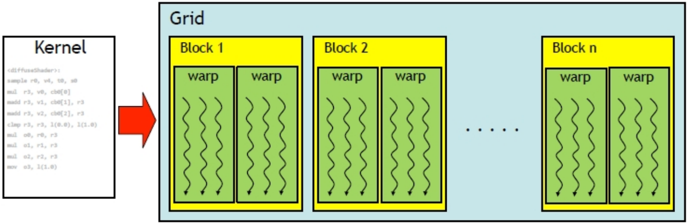
ALU: Arithmetic Logic Unit (thing that does calculations!)

CPU: Lots of fast memory (cache), few ALUs

GPU: Little fast memory, lots of ALUs

Some definitions

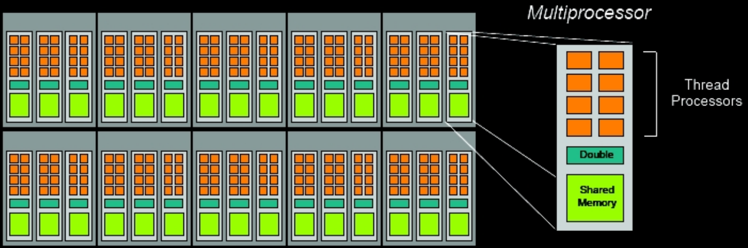
- Kernel
  - GPU program that runs on a thread grid
- Thread hierarchy
  - Grid : a set of blocks
  - Block : a set of warps
  - Warp : a SIMD group of 32 threads
  - Grid size * block size = total # of threads



Credit: CS264 (N. Pinto)

10-Series Architecture

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



NVIDIA

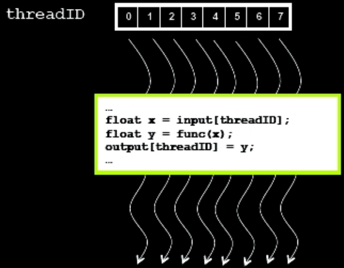
CUDA Kernels and Threads

- Parallel portions of an application are executed on the device as kernels
  - One kernel is executed at a time
  - Many threads execute each kernel
- Differences between CUDA and CPU threads
  - CUDA threads are extremely lightweight
    - Very little creation overhead
    - Instant switching
  - CUDA uses 1000s of threads to achieve efficiency
    - Multi-core CPUs can use only a few

Definitions
Device = GPU
Host = CPU
Kernel = function that runs on the device

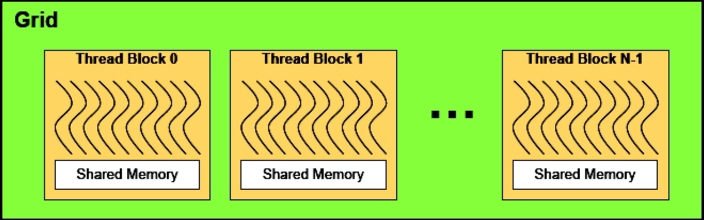
Arrays of Parallel Threads

- A CUDA kernel is executed by an array of threads
  - All threads run the same code
  - Each thread has an ID that it uses to compute memory addresses and make control decisions



Thread Batching

- Kernel launches a grid of thread blocks
  - Threads within a block cooperate via shared memory
  - Threads within a block can synchronize
  - Threads in different blocks cannot cooperate
- Allows programs to *transparently scale* to different GPUs



Low-Level Programming for GPU's

– Languages:

- CUDA :: http://www.nvidia.com/object/cuda_home_new.html
- OpenCL :: http://www.khronos.org/opencl/

- Which? http://wiki.tiker.net/CudaVsOpenCL

About CUDA

CUDA is. . .

About CUDA

CUDA is...

- a bunch of C/C++ libraries allowing the coder to use the GPU

About CUDA

CUDA is. . .

- a bunch of C/C++ libraries allowing the coder to use the GPU
- a fine-grain, low-level language (user controls all memory management, synchronicity etc)

About CUDA

CUDA is. . .

- a bunch of C/C++ libraries allowing the coder to use the GPU
- a fine-grain, low-level language (user controls all memory management, synchronicity etc)
- for NVIDIA GPU's only (will not work on AMD GPU's)

About CUDA

CUDA is. . .

- a bunch of C/C++ libraries allowing the coder to use the GPU
- a fine-grain, low-level language (user controls all memory management, synchronicity etc)
- for NVIDIA GPU's only (will not work on AMD GPU's)

About CUDA

CUDA is...

- a bunch of C/C++ libraries allowing the coder to use the GPU
- a fine-grain, low-level language (user controls all memory management, synchronicity etc)
- for NVIDIA GPU's only (will not work on AMD GPU's)

There are also new higher-level interfaces to CUDA that do much of the dirty work for you...

Example CUDA Program

My example, modified from some code on the NVIDIA forums:

See CUDA_example.cu

Compile with:

```
nvcc CUDA_example.cu -use_fast_math -o cosine.out
```

Run with:

```
./cosine.out
```

GPU-accelerated Libraries

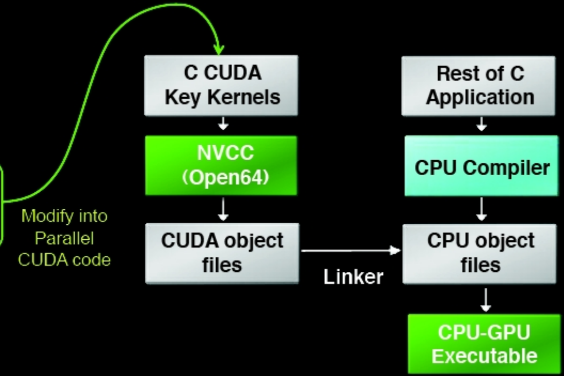
- Thrust (C++ STL-type library)
- CULA (CUDA implementation of LAPACK and BLAS, dense & sparse by Photonics)
- cuBLAS (CUDA implementation of BLAS by NVIDIA)
- cuSPARSE (CUDA implementation for sparse matrices by NVIDIA)
- cuRAND (CUDA random number generation by NVIDIA)
- CUDA Math Library (by NVIDIA)

Other Interfaces to GPUs

- PyCUDA :: `http://documen.tician.de/pycuda/`
- PyOpenCL :: `http://documen.tician.de/pyopencl/`
- R Packages:
  - `gputools`
- OpenACC (essentially an OpenMP for GPU's)
- Other?

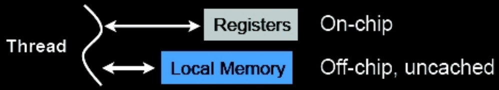
Compiling C with CUDA Applications

```
void serial_function(… ) {
  ...
}
void other_function(int ... ) {
  ...
}

void saxpy_serial(float ... ) {
   for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```



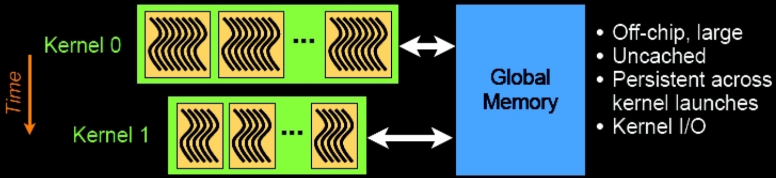
Kernel Memory Access

- **Per-thread**



- **Per-block**

- **Per-device**



CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
`int var;`	register	thread	thread
`int array_var[10];`	local	thread	thread
`__shared__ int shared_var;`	shared	block	block
`__device__ int global_var;`	global	grid	application
`__constant__ int constant_var;`	constant	grid	application

- "automatic" scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- "automatic" array variables without qualifier reside in thread-local memory

CUDA Variable Type Performance

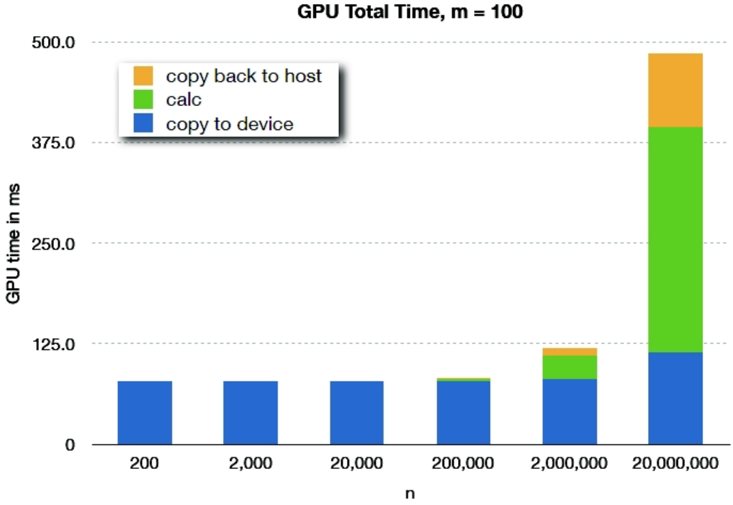
Variable declaration	Memory	Penalty
`int var;`	register	1x
`int array_var[10];`	local	100x
`__shared__ int shared_var;`	shared	1x
`__device__ int global_var;`	global	100x
`__constant__ int constant_var;`	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

CUDA Variable Type Scale

Variable declaration		Instances	Visibility
	`int var;`	100,000s	1
	`int array_var[10];`	100,000s	1
`__shared__`	`int shared_var;`	100s	100s
`__device__`	`int global_var;`	1	100,000s
`__constant__`	`int constant_var;`	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads



Credit: CS264 (N. Pinto)

Perspective on GPU's

What tasks are they good for?

Perspective on GPU's

What tasks are they good for?

- ☺ Numerical integration (nearly always)
- ☺ (Very) slow iteration MCMC (use within-iteration parallelism)
- ☺ 'Simple' bootstraps
- ☺ Particle Filtering (Sequential Monte Carlo)
- ☺ (Extremely difficult) brute force optimization
- ☺ (Very) Large matrix calculations
- ☺ Single-use applications

Perspective on GPU's

What tasks are they not good for?

Perspective on GPU's

What tasks are they not good for?

- ☹ Fast iteration MCMC
- ☹ 'Difficult' bootstraps
- ☹ (Most) optimization problems
- ☹ Methodological work (portable code)
- ☹ Any problem that is not worth the additional effort...

Resources

- http://www.cs264.org/
- http://www.nvidia.com/object/cuda_home_new.html
- http://developer.nvidia.com/cuda-downloads
- http://developer.nvidia.com/nvidia-gpu-computing-documentation
- http://developer.nvidia.com/cuda-training#2
- http://developer.nvidia.com/getting-started-parallel-computing

Getting started:

- Find a CUDA-enabled computer and install CUDA first!
- NVIDIA GPU Computing SDK has lots of (rich) examples
- Courses found above have lots of nice labs

Appendix: Installation (Mac & Linux)

- Install the CUDA driver
- Install the CUDA Toolkit (sets up compiler, libraries etc.)
- Add environment variables to ~/.bash_profile:

```
export PATH=/usr/local/cuda/bin:$PATH
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

- Install the CPU Computing SDK (lots of code examples)
- Verify the install with:

```
kextstat | grep -i cuda
nvcc -V
cd /Developer/GPU\ Computing/C/bin/darwin/release
./deviceQuery
```

CUDA on my Macbook Pro (10.6.8)

```
Device 0: "GeForce 320M"
  CUDA Driver Version / Runtime Version          4.1 / 4.1
  CUDA Capability Major/Minor version number:    1.2
  Total amount of global memory:                 253 MBytes (265027584 bytes)
  ( 6) Multiprocessors x ( 8) CUDA Cores/MP:     48 CUDA Cores
  GPU Clock Speed:                               0.95 GHz
  Memory Clock rate:                             1064.00 Mhz
  Memory Bus Width:                              128-bit
  Max Texture Dimension Size (x,y,z)             1D=(8192), 2D=(65536,32768), 3D=(2048,2048,2048)
  Max Layered Texture Size (dim) x layers        1D=(8192) x 512, 2D=(8192,8192) x 512
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       16384 bytes
  Total number of registers available per block: 16384
  Warp size:                                     32
  Maximum number of threads per block:           512
  Maximum sizes of each dimension of a block:    512 x 512 x 64
  Maximum sizes of each dimension of a grid:     65535 x 65535 x 1
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             256 bytes
  Concurrent copy and execution:                 Yes with 1 copy engine(s)
...
deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 4.1, CUDA Runtime Version = 4.1, NumDevs = 1, Dev
```

Testing CUDA

```
./bandwidthTest

 Device 0: GeForce 320M
 Quick Mode

 Host to Device Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes) Bandwidth(MB/s)
   33554432 581.1

 Device to Host Bandwidth, 1 Device(s), Paged memory
   Transfer Size (Bytes) Bandwidth(MB/s)
   33554432 609.9

 Device to Device Bandwidth, 1 Device(s)
   Transfer Size (Bytes) Bandwidth(MB/s)
   33554432 5965.2

[bandwidthTest] test results...
PASSED
```