Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R
Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R
The single instruction, multiple data (SIMD) paradigm

- SIMD: apply the same command to multiple places in a dataset.

```c
for (i = 0; i < 1e6; ++i )
    a[i] = b[i] + c[i];
```

- On CPUs, the iterations of the loop run sequentially.
- With GPUs, we can easily run all 1,000,000 iterations simultaneously.

```c
i = threadIdx.x;
    a[i] = b[i] + c[i];
```

- We can similarly parallelize a lot more than just loops.
## Parallel MCMC by Lee, Yau, Giles, and others

<table>
<thead>
<tr>
<th># chains</th>
<th>CPU time (min)</th>
<th>GTX 280 (min)</th>
<th>CPU time / GPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>0.0166</td>
<td>0.0148</td>
<td>1.1</td>
</tr>
<tr>
<td>32</td>
<td>0.0656</td>
<td>0.0151</td>
<td>4</td>
</tr>
<tr>
<td>128</td>
<td>0.262</td>
<td>0.0154</td>
<td>17</td>
</tr>
<tr>
<td>512</td>
<td>1.04</td>
<td>0.0174</td>
<td>60</td>
</tr>
<tr>
<td>2,048</td>
<td>4.16</td>
<td>0.0248</td>
<td>168</td>
</tr>
<tr>
<td>8,192</td>
<td>16.64</td>
<td>0.0720</td>
<td>230</td>
</tr>
<tr>
<td>32,768</td>
<td>66.7</td>
<td>0.249</td>
<td>268</td>
</tr>
<tr>
<td>131,072</td>
<td>270.3</td>
<td>0.970</td>
<td>279</td>
</tr>
</tbody>
</table>
## Parallel sequential MC by Lee, Yau, Giles, and others

<table>
<thead>
<tr>
<th>Sample size</th>
<th>CPU time (min)</th>
<th>GTX 280 (min)</th>
<th>CPU time / GPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>8,192</td>
<td>4.44</td>
<td>0.0199</td>
<td>223.1</td>
</tr>
<tr>
<td>16,384</td>
<td>8.82</td>
<td>0.0355</td>
<td>263</td>
</tr>
<tr>
<td>32,768</td>
<td>17.7</td>
<td>0.0666</td>
<td>265</td>
</tr>
<tr>
<td>65,536</td>
<td>35.3</td>
<td>0.131</td>
<td>269</td>
</tr>
<tr>
<td>131,076</td>
<td>70.6</td>
<td>0.261</td>
<td>270.5</td>
</tr>
<tr>
<td>262,144</td>
<td>141</td>
<td>0.52</td>
<td>271.2</td>
</tr>
</tbody>
</table>
Parallel Bayesian EM by Suchard, Wang, Chan, and others

<table>
<thead>
<tr>
<th>Sample size</th>
<th>cpu 1 (sec)</th>
<th>gpu 1 (sec)</th>
<th>CPU time / GPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^2$</td>
<td>4.0</td>
<td>71.1</td>
<td>0.1</td>
</tr>
<tr>
<td>$10^3$</td>
<td>40.0</td>
<td>81.3</td>
<td>0.5</td>
</tr>
<tr>
<td>$10^4$</td>
<td>607.0</td>
<td>91.2</td>
<td>6.7</td>
</tr>
<tr>
<td>$10^5$</td>
<td>7793.0</td>
<td>129.6</td>
<td>60.1</td>
</tr>
<tr>
<td>$10^6$</td>
<td>78765.0</td>
<td>680.6</td>
<td>115.7</td>
</tr>
</tbody>
</table>
Other applications

- Clustering
- Bootstrap
- Regression
- Matrix algebra
- EM Algorithm
- Rejection sampling
- Multiple testing
- Cross validation
- ...

Will Landau (Iowa State University)
Computer processors

- **Processing unit**: a computer chip that performs executive functions.

- **Core**: One of possibly many “sub-processors” placed on the same processing unit, each of which has the full functionality of the processing unit.
The Central Processing Unit (CPU)

- Regular computer processor.
- Allows parallelism, but not massive parallelism on its own.
- Usually contains 1 to 8 cores.
- Examples:
  - Intel 8086 (1979, x86)
  - Intel Core 2 Duo
  - Intel 80486DX2 (below)
The Graphics Processing Unit (GPU)

- Processor in a video card or graphics card.
- Massively parallel: originally designed to speed up graphics throughput in video games.
- Cannot run by itself. Needs to be hooked up to a computer with a CPU.
- Contains several hundred cores.
- Higher memory bandwidth than a CPU.
- Examples:
  - NVIDIA GeForce 6600 (bottom left)
  - NVIDIA GeForce GTX 580
  - NVIDIA Tesla M2070 (on our GPU-enabled machines)
CPU / GPU cooperation

- The CPU ("host") is in charge.
- The CPU sends computationally intensive instruction sets to the GPU ("device") just like a human uses a pocket calculator.

![Diagram of CPU and GPU cooperation](image)
More on parallelism

- **Parallelism**: the practice of running multiple calculations simultaneously.
- The architecture of GPUs is extremely well-suited to massively parallel workflows.
- Note: GPU parallelism is one of many kinds of parallelism. Others include:
  - Posix threads (CPU parallelism)
  - parallel cloud computing
  - openMP parallelism
  - openMP parallelism
GPU parallelism speeds up calculations

- Amdahl’s Law says that the maximum theoretical speedup (CPU time / GPU time) is

\[ \frac{1}{1 - P + \frac{P}{N}} \]

where:
- \( P = \) fraction of the program (in terms of execution time) that can be parallelized
- \( N = \) number of parallel processors

- As \( N \rightarrow \infty \), Amdahl’s quantity goes to

\[ \frac{1}{1 - P} \]

- So if 99% of the program can be parallelized, we could theoretically have a 100-fold speedup.
How GPU parallelism works

1. The CPU sends a command called a **kernel** to a GPU.
2. The GPU executes several duplicate realizations of this command, called **threads**.
   - These threads are grouped into bunches called **blocks**.
   - The sum total of all threads in a kernel is called a **grid**.

Toy example:
- CPU says something like, “Hey, GPU. Sum pairs of adjacent numbers. Use the array, (1, 2, 3, 4, 5, 6, 7, 8). Use 2 blocks of 2 threads each.”
- GPU thinks: “Sum pairs of adjacent numbers” is a kernel that I need to apply to the array, (1, 2, 3, 4, 5, 6, 8).
- The GPU spawns 2 blocks, each with 2 threads:

<table>
<thead>
<tr>
<th>Block</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Action</td>
<td>1+2</td>
<td>3+4</td>
</tr>
</tbody>
</table>

- All four actions above happen simultaneously.
- I could have also used 1 block with 4 threads and given the threads different pairs of numbers.
Introduction to GPU computing for statisticians

Will Landau

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R
Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R
CUDA: making a gaming toy do science

- GPUs were originally meant to speed up graphical displays for Windows OS and video games.

- **CUDA**: Compute Unified Device Architecture.
- Before CUDA, programmers could only program on GPUs using graphics languages, which are appropriate for video games but clumsy for science.
- CUDA devices support CUDA C, an extension of C for programs that use GPUs.
CUDA-enabled servers at Iowa State

- impact1.stat.iastate.edu (Red Hat Enterprise Linux Server release 6.2)
- impact2.stat.iastate.edu (CentOS release 6.3)
- impact3.stat.iastate.edu (Red Hat Enterprise Linux Server release 6.4)
- impact4.stat.iastate.edu (CentOS release 6.4)
Specs of our CUDA systems

- No graphical user interface or remote desktop capabilities. (Use the Linux command line.)
- 24 CPUs and 4 Tesla M2070 GPUs, where each GPU has 448 cores:
- For more specs, log into impact1, 2, or 3 and enter into the command line:

```
1 cd /usr/local/NVIDIA_GPU_Computing.SDK/C/bin
   /linux/release
2 ./deviceQuery
```
Logging in

- Open a command line program (Terminal in Linux and Mac OS, Cygwin or MinGW in Windows).
- Enter:

```
1  ssh -p 323 ISU_ID@impact1.stat.iastate.edu
```

- Note: in general, the port number for ssh is not always 323.
- Refer to http://www.linuxcommand.org/ or contact me at landau@iastate.edu for help with the Linux command line.
- Contact Stat IT at statit@iastate.edu or me if:
  - You can’t log on, or
  - You want to be able to log on without entering your password every time, or
  - You want to shorten ssh -p 323 ISU_ID@impact1.stat.iastate.edu into a more manageable alias on your local machine.
Important directories

- **/home/ISU_ID** Your private home folder on SMB (the collective storage system for all the stat dept linux servers). Files in here are stored remotely on SMB, not locally on impact1-3.

- **/Cyfiles/ISU_ID** Your private Cyfiles folder. Files in here are stored remotely on the Cyfiles server, not locally on impact1-3.

- **/tmp**
  - Everything in here is stored locally on impact1, etc., wherever you’re logged in.
  - To avoid communicating over a network when you want fast computation, put large datasets here.
  - Note: **/tmp** automatically empties periodically.

- **/usr/local/NVIDIA_GPU_Computing_SDK**
  - Example CUDA C code. Stored locally on impact1, etc.
  - You do not have write privileges here.
Outline

- GPUs, parallelism, and why we care
- CUDA and our CUDA systems
- GPU computing with R
GPU-enabled R packages

- **WideLM** - used to quickly fit a large number of linear models to a fixed design matrix and response vector.
- **magma** - a small linear algebra with implementations of backsolving and the LU factorization.
- **cudaBayesreg** - implements a Bayesian model for fitting fMRI data.
- **gcbd** - a Debian package for benchmarking linear algebra algorithms such as the QR, SVD and LU.factorizations.
- **gputools** - probably the most useful of these 5.
## Contents of gputools

### Choose your device:

<table>
<thead>
<tr>
<th>gputools function</th>
<th>CPU analog</th>
<th>Same usage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>chooseGpu()</td>
<td>none</td>
<td>NA</td>
</tr>
<tr>
<td>getGpuId()</td>
<td>none</td>
<td>NA</td>
</tr>
</tbody>
</table>

### Linear algebra:

<table>
<thead>
<tr>
<th>gputools function</th>
<th>CPU analog</th>
<th>Same usage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpuDist()</td>
<td>dist()</td>
<td>no</td>
</tr>
<tr>
<td>gpuMatMult()</td>
<td>%*% operator</td>
<td>no</td>
</tr>
<tr>
<td>gpuCrossprod()</td>
<td>crossprod()</td>
<td>yes</td>
</tr>
<tr>
<td>gpuTcrossprod()</td>
<td>tcrossprod()</td>
<td>yes</td>
</tr>
<tr>
<td>gpuQr()</td>
<td>qr()</td>
<td>almost</td>
</tr>
<tr>
<td>gpuSolve()</td>
<td>solve()</td>
<td>no</td>
</tr>
<tr>
<td>gpuSvd()</td>
<td>svd()</td>
<td>almost</td>
</tr>
</tbody>
</table>
### Simple model fitting:

<table>
<thead>
<tr>
<th>gputools function</th>
<th>CPU analog</th>
<th>Same usage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpuLm()</td>
<td>lm()</td>
<td>yes</td>
</tr>
<tr>
<td>gpuLsfit()</td>
<td>lsfit()</td>
<td>yes</td>
</tr>
<tr>
<td>gpuGlm()</td>
<td>glm()</td>
<td>yes</td>
</tr>
<tr>
<td>gpuGlm.fit()</td>
<td>glm.fit()</td>
<td>yes</td>
</tr>
</tbody>
</table>

### Hypothesis testing:

<table>
<thead>
<tr>
<th>gputools function</th>
<th>CPU analog</th>
<th>Same usage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpuTtest()</td>
<td>t.test()</td>
<td>no</td>
</tr>
<tr>
<td>getAucEstimate()</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>
### Other routines:

<table>
<thead>
<tr>
<th>gputools function</th>
<th>CPU analog</th>
<th>Same usage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>gpuHclust()</td>
<td>hclust()</td>
<td>no</td>
</tr>
<tr>
<td>gpuDistClust()</td>
<td>hclust(dist())</td>
<td>no</td>
</tr>
<tr>
<td>gpuFastICA()</td>
<td>fastICA() (fastICA package)</td>
<td>yes</td>
</tr>
<tr>
<td>gpuGranger()</td>
<td>grangertest() (lmtest package)</td>
<td>no</td>
</tr>
<tr>
<td>gpuMi()</td>
<td>???</td>
<td>???</td>
</tr>
<tr>
<td>gpuSvmPredict()</td>
<td><a href="http://www.jstatsoft.org/v15/i09/paper">www.jstatsoft.org/v15/i09/paper</a></td>
<td>no</td>
</tr>
<tr>
<td>gpuSvmTrain()</td>
<td><a href="http://www.jstatsoft.org/v15/i09/paper">www.jstatsoft.org/v15/i09/paper</a></td>
<td>no</td>
</tr>
</tbody>
</table>
Example

```
1 > getGpuID()
2 [1] 0
3 > chooseGpu(3)
4 [[1]]
5 [1] 3
6 > getGpuID()
7 [1] 3
8 > A <- matrix(runif(1e7), nrow = 1e4)
9 > B <- matrix(runif(1e7), nrow = 1e4)
10 > ptm <- proc.time(); C <- gpuMatMult(A, B);
11 > proc.time() - ptm
12           user   system elapsed
13        2.959  2.190   5.159
14 > ptm <- proc.time(); C <- A %*% B;
15 > proc.time() - ptm
16           user   system elapsed
17   116.389  0.166  116.503
```
### Speedup

The figure illustrates the speedup of `glm()` vs `gpuGlm()` in R. The total scheduled runtime (seconds) is plotted on the y-axis against the base 10 log of the number of observations. The chart shows:

- **Mean CPU runtime** indicated by blue dots.
- **Mean GPU runtime** indicated by green dots.
- **First GPU run (overhead, discarded from confidence region calculations)** indicated by red dots.

The graph highlights the significant speedup achieved with GPU computing, particularly as the number of observations increases.
Introduction to GPU computing for statisticians

Will Landau

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R

Speedup

**Fig. 2: Granger Times**

![Graph showing execution times for GPU and CPU](image)

**Execution Time (secs)**

- **GPU**
- **CPU**

**Num. Rand. Vars. (10 obs. each, lag 2)**

- 200
- 400
- 600
- 800
- 1000

- 0

Speedup
Tentative Syllabus

1. Intro and gputools
2. A codeless intro to GPU parallelism
3. Intro to CUDA C
4. CUDA C: K-means and MCMC
5. CUDA C: Shared memory and performance measurement
6. CUDA C: Race conditions, atomics, and warps
7. CUBLAS and CULA: linear algebra libraries for CUDA C
8. CURAND: a GPU-enabled library for fast random number generation
10. Intro to Python: preparation for PyCUDA
11. PyCUDA: a Python module for GPU computing
Outline

GPUs, parallelism, and why we care

CUDA and our CUDA systems

GPU computing with R
Resources


That’s all for today.

- Series materials are available at http://will-landau.com/gpu.