The Practical Reality of Heterogeneous Super Computing

Rob Farber
Visiting scientist at the NVIDIA CUDA Research Center at the Irish Center for High-End Computing (ICHEC)
Outline of the talk

• CUDA is now a language for all application development just like C/C++ and Java!
• Strategies for embracing heterogeneous computing
  – Opportunities enabled by CUDA x86
  – Practical ideas for balancing CPU & GPU
  – Practical tips on running CUDA Kernels on CPU cores
The growth of CUDA

• First introduced in February 2007
  – Now taught at 454 institutions world-wide
Performance is the reason for GPUs

Top 100 NVIDIA CUDA application showcase speedups as of May, 9, 2011
(Min 100, Max 2600, Median 1350)

Reported speedup

Ranked from highest to lowest speedup

**Why x86? (Why ARM?) (Why ...?)**

- **Market accessibility:**
  - Over \( \frac{1}{4} \) BILLION CUDA-enabled GPUs sold (300M)
  - Small compared to the number of x86 systems.

- **ARM is the power behind many super phones**
  - What a market segment! (cellphones, tablets, ...)

---

**Performance is on a Log scale**

A 3 Watt Kal-El is 5x a Tegra 2

One a year roadmap

---

Core2Duo
“CUDA is for GPUs and CPUs! “

“One source tree to hold them all and on the GPU accelerate them!” (A parody of J.R.R. Tolkien)
A convergence of concepts (CPU 2-6 cores/GPU hundreds of cores)
CUDA is no longer just for GPUs

- NVIDIA's nvcc
- Ocelot (PTX to x86 emulation and translation)
- MCUDA (CUDA to C translation)
- SWAN (CUDA to OpenCL translation)

- PGI CUDA-x86
- NVIDIA GPU
- X86_64 CPU
- AMD GPU
- OpenCL
PGI deviceQuery on a Xeon e5560

CUDA Device Query (Runtime API) version (CUDART static linking)

There is 1 device supporting CUDA

Device 0: "DEVICE EMULATION MODE"
CUDA Driver Version: 99.99
CUDA Runtime Version: 99.99
CUDA Capability Major revision number: 9998
CUDA Capability Minor revision number: 9998
Total amount of global memory: 128000000 bytes
Number of multiprocessors: 1
Number of cores: 0
Total amount of constant memory: 1021585952 bytes
Total amount of shared memory per block: 1021586048 bytes
Total number of registers available per block: 1021585904
Warp size: 1
Maximum number of threads per block: 1021585920
Maximum sizes of each dimension of a block: 32767 x 2 x 0
Maximum sizes of each dimension of a grid: 1021586032 x 32767 x 1021586048
Maximum memory pitch: 4206313 bytes
Texture alignment: 1021585952 bytes
Clock rate: 0.00 GHz
Concurrent copy and execution: Yes
Run time limit on kernels: Yes
Integrated: No
Support host page-locked memory mapping: Yes
Compute mode: Unknown
Concurrent kernel execution: Yes
Device has ECC support enabled: Yes


PASSED

Press <Enter> to Quit...
PGI running `arrayReversal_multiblock_fast.cu` from Part 3 of my DDJ tutorials

Correct!

It just compiles and runs:
- Boring from a presentation point of view.
- **Exciting from an opportunity point of view.**
PGI to ship a unified binary in 2012
Ocelot runs CUDA binaries

- Must install Ocelot
- Offers a lot more than just x86
  - Profiling/hot-spotting
- Not a turn-key system!
Thrust: CUDA made simple

- Most of the actual code from an example that scales to 500 GPUs and delivers 341-times speedup over a single-core (32-bit) Xeon CPU.

```cpp
FCnOfInterest objFcn(input);

energy = thrust::transform_reduce(
    thrust::counting_iterator<int>(0),
    thrust::counting_iterator<int>(nExamples),
    objFcn,
    0.0f,
    thrust::plus<Real>())
```
Functors can run on both the host and device

```cpp
__device__ __host__
Real operator()(unsigned int tid) {
    const register Real* in = &examples[tid * exLen];
    register int index=0;
    register Real h1 = p[index++];
    register Real o = p[index++];

    h1 += in[0] * p[index++];
    h1 += in[1] * p[index++];
    h1 = G(h1);

    o += in[0] * p[index++];
    o += in[1] * p[index++];
    o += h1 * p[index++];

    // calculate the square of the diffs
    o -= in[nInput];
    return o * o;
}
```
Thrust can use an OpenMP backend

nvcc -O2 -o monte_carlo monte_carlo.cu -Xcompiler -fopenmp \ 
-DTHRUST_DEVICE_BACKEND=THRUST_DEVICE_BACKEND_OMP \ 
-lcudart -lgomp

<table>
<thead>
<tr>
<th>Device</th>
<th>seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>0.222</td>
</tr>
<tr>
<td>4 OpenMP threads</td>
<td>2.090</td>
</tr>
<tr>
<td>2 OpenMP threads</td>
<td>4.168</td>
</tr>
<tr>
<td>1 OpenMP thread</td>
<td>8.333</td>
</tr>
</tbody>
</table>

- Timing reported on the Thrust website show that the performance is acceptable
- Be aware that Thrust is not optimized to produce the best x86 runtimes
Strategies for embracing heterogeneous computing.

– Opportunities enabled by CUDA x86
– Practical ideas for balancing CPU & GPU
– Practical tips on running CUDA Kernels on CPU cores
The *one CUDA source tree* rationale (aside from saving development $)

- **Fast**: A compiler can perform optimizations that a PTX-based system like Ocelot will miss
  – Please prove me wrong!

- **Transparent**: Both NVIDIA and PGI state that even CUDA applications utilizing proprietary features of the GPU texture units will exhibit identical behavior on both x86 and GPU hardware
  – Please don’t prove me wrong!

- **Convenient**: ship one binary to customers for GPU and x86
Reasons for CUDA for all apps

1. Not much of a change for many applications and organizations
   a. CUDA is based on standard C and C++
   b. Both of these languages have a solid history of application development spanning decades

2. Makes applications faster
   a. CUDA gives the programmer the ability to better exploit parallelism
   b. Exploit the SIMD parallelism in the AVX or SSE instruction in each x86 core
Reasons for CUDA for all apps

3. Helps to avoid parallel bugs:
   a. The CUDA execution model precludes common parallel programming errors including race conditions and deadlock
      • Programmer still has to update shared memory correctly

4. A growing tool ecosystem
   a. cuda-gdb/Parallel Nsight can debug massively parallel apps with large # of concurrent operations
   b. NVIDIA: Parallel Nsight, computefprof
   c. Others: TAU/PAPI profiler, Ocelot
Reasons for CUDA for all apps

5. Scalability of the model:
   a. 100k threads = no big deal, (1M threads = ...), (...)
   b. Save future software development dollars and allow fast penetration into new markets and technology platforms

6. GPU acceleration comes for free
   a. Opens the door for order of magnitude application acceleration
   b. Expands market reach to the ¼ billion CUDA-enabled GPUs that have been sold worldwide
   c. Future-proofs applications
Reasons for CUDA for all apps

7. There are many CUDA developers
   a. This developer base is rapidly expanding
   b. CUDA is currently taught at over 454 universities and colleges worldwide -> also rapidly expanding
      • ICHEC is in the final stages of becoming a CUDA Teaching Center
Strategies for embracing heterogeneous computing

- Opportunities enabled by CUDA x86
- Practical ideas for balancing CPU & GPUs
- Practical tips on running CUDA Kernels on CPU cores
Three rules for fast GPU codes

1. Get the data on the GPU (and keep it there!)
   - PCIe x16 v2.0 bus: 8 GiB/s in a single direction
   - Compute 2.0/2.1 GPUs: 140-200 GiB/s

2. Give the GPU enough work to do
   - Assume 10 μs latency and 1 TF device
   - Can waste \((10^{-6} \times 10^{12}) = 1M\) operations

3. Reuse and locate data to avoid global memory bandwidth bottlenecks
   - \(10^{12}\) flop hardware delivers \(10^{10}\) flop when global memory limited
   - Can cause a 100x slowdown!

Corollary: Avoid malloc/free!
Data movement still happens on x86
PGI bandwidthTest on a Xeon e5560

Running on...

Device 0: DEVICE EMULATION MODE
Quick Mode

Host to Device Bandwidth, 1 Device(s), Paged memory
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                4152.5

Device to Host Bandwidth, 1 Device(s), Paged memory
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                4257.0

Device to Device Bandwidth, 1 Device(s)
Transfer Size (Bytes)    Bandwidth(MB/s)
33554432                8459.2

[bandwidthTest] - Test results:
PASSED

Happens with straight compilation of CUDA codes
• PGI allows x86 programs to just do a pointer assignment
Heterogeneous apps with CUDA libraries

Ivan Girotto and Filippo Spiga ICHEC

- An important example: electronic-structure calculations and materials modeling at the nanoscale
- SCF* calculation on plane wave (64-bit calculations)
- Main bottlenecks
  - 3D FFT (-> CUFFT)
  - Linear Algebra
    - Matrix Matrix Multiplication (-> CUBLAS)
    - Eigenvalues and Eigenvectors (work in-progress)
- First GPU-enabled beta released on May 2011

* Plane-Wave Gaussian Self-Consistent Field Method
A good start: 8-times speedup over serial

Mixing CPU and GPU threads using OpenMP(PWscf AUSURF112)

http://www.quantum-espresso.org/

112 atom simulation

* Shorter bar means less walltime
Existing work Dgemm for Linpack HPL


**DGEMM: C = alpha A B + beta C**

\[
DGEMM(A, B, C) = DGEMM(A, B_1, C_1) \cup DGEMM(A, B_2, C_2)
\]

The idea can be extended to multi-GPU configuration and to handle huge matrices.

Find the optimal split, knowing the relative performances of the GPU and CPU cores on DGEMM.
Watch out for PCIe configuration!
(and benchmarkman’s ship!)

Two GPU CUFFT run (some benchmarks use individual PCIe buses)
Current leader: CPU+GPU 435 GF/s


---

**Fermi DGEMM Performance**

- **435 CPU+GPU**
- **350 GPU**
- **85 CPU**

- Dual Quad-Core Xeon X5550 2.66 GHz 8 cores MKL 10.2.4.032
- Tesla M2050 "Fermi" 1.15 GHz
- CPU + GPU

Size: N=M (K=1024)
The phiGEMM library from ICHEC

• A library that you use like CUBLAS
  – Transparently manages the thunking operations
  – Supports Sgemm(), Dgemm(), and Zgemm()
  – Asynchronous data transfer (via PINNED Memory)
  – MultiGPU management through single process (CUDA 4.0)

• Evolving: Possible improvements via multi-stream out-of-order execution (see http://www.nvidia.com/content/GTC-2010/pdfs/2057_GTC2010.pdf)

• Written by Girotto and Spiga. Freely downloadable from http://qe-forge.org/projects/phigemm/
phiGEMM matches 1 GPU performance

2 x Intel Xeon X5680 3.33GHz + NVIDIA Tesla C2050

2 x Intel Xeon X5680 3.33GHz

System provided by

H2D = ~ 5.5GB/s
D2H = ~ 6.0GB/s

MKL + CUBLAS theoretical peak

MKL + CUBLAS (PEAK)

MKL

THUNKING CUBLAS

MKL

CUBLAS (PEAK)

phiGEMM

http://qe-forge.org/projects/phigemm/
phiGEMM dual GPU/single bus

http://qe-forgo.org/projects/phigemm/

2 x Intel Xeon X5680 3.33GHz + 2 NVIDIA Tesla C2050

System provided by

H2D = ~ 2.8GB/s  D2H = ~ 3.2GB/s

M = N = K (DP Size)
phiGEMM dual GPU/dual bus

http://qe-forge.org/projects/phigemm/

2 x Intel Xeon X5680 3.33GHz + 2 NVIDIA Tesla C2050

System provided by

MKL + CUBLAS Peak

Faster here

Thunking CUBLAS

GPU0
- H2D = ~ 4.8GB/s
- D2H = ~ 5.0GB/s

GPU1
- H2D = ~ 4.3GB/s
- D2H = ~ 4.8GB/s

MKL

M = N = K (DP Size)

GFLOPS

1024 2048 3072 4096 5120 6144 7168 8192 9216 10240

MKL

THUNKING CUBLAS

CUBLAS (PEAK)

phiGEMM

MKL + CUBLAS (PEAK)
Performance is dependent on problem size

- phiGEMM can run GEMM on matrices that do not fit on a single GPU
- Recursive call to phiGEMM with smaller sub-matrix
BIG phiGEMM multi GPU/single bus

http://qe-forge.org/projects/phigemm/

2 x Intel Xeon X5670 2.93GHz + 4 NVIDIA Tesla C2050

System provided by NVIDIA

\[ M = K = N = 25000 \text{ (DP)} = 15\text{GBytes} \]

<table>
<thead>
<tr>
<th>GPU Configuration</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1GPU</td>
<td>277</td>
</tr>
<tr>
<td>2GPUs</td>
<td>522 \times 1.9</td>
</tr>
<tr>
<td>4GPUs</td>
<td>809 \times 2.9</td>
</tr>
</tbody>
</table>

\[ x \text{ x } 1.9 \text{ x } 2.9 \]
BIG phiGEMM multi GPU/dual bus

2 x Intel Xeon X5670 2.93GHz + 4 NVIDIA Tesla C2050

M = K = N = 25000 (DP) = 15GBytes

277 x 2.0 = 551
551 x 3.4 = 942

GFLOPS

GPU

CPU

System provided by

http://qe-forge.org/projects/phigemm/
**Gemm operations are compute intensive**

*Gemm is a Level 3 BLAS operation: Work per datum transferred is high \(O(N)\)

<table>
<thead>
<tr>
<th>BLAS level</th>
<th>Data</th>
<th>Work</th>
<th>Work per Datum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(O(N))</td>
<td>(O(N))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>2</td>
<td>(O(N^2))</td>
<td>(O(N^2))</td>
<td>(O(1))</td>
</tr>
<tr>
<td>3</td>
<td>(O(N^2))</td>
<td>(O(N^3))</td>
<td>(O(N))</td>
</tr>
</tbody>
</table>

Let’s look at a problem that is more dependent on data transfers: 3D FFTs
Performance 3DFFT on multi-GPU

WALL TIME (seconds)

- 64^3 x 4096 (Times)
- 128^3 x 512 (Times)
- 512^3 x 32 (Times)

1GPU vs 2GPUs (1Bus) vs 4GPUs (1Bus)

- ~1.8
- ~2.4
- ~1.8
- ~2.4
- ~1.8
- ~2.4
Performance 3DFFT on multi-GPU

Single 3DFFT on GPU Vs FFTW3 (fftw_plan_many_dft)

WALL TIME (seconds)

- 1GPU
- 2GPUs (1Bus)
- 4GPUs (1Bus)
- 1CPU

- 64^3 x 4096 (Times)
- 128^3 x 512 (Times)
- 512^3 x 32 (Times)
Lessons learned

- Watch out for shortcuts with the PCIe bus!
- Thunking can deliver high performance
- Libraries like phiGEMM can make multiGPU/hybrid application development transparent and compatible with libraries like CUBLAS
- I envision a multi/hybrid “smart pointer” to create a non-thunking interface
  - Rule 1: Get the data on the GPU and keep it there
ICHEC contribution to MAGMA

- Like MAGMA, phiGEMM aims "to design linear algebra algorithms and frameworks for hybrid manycore and GPUs systems that can enable applications to fully exploit the power that each of the hybrid components offers."
  - Quote from the MAGMA website (http://icl.cs.utk.edu/magma/)

- phiGEMM is under consideration for inclusion in the MAGMA library
Really Exciting! Hybrid Codes

• MAGMA (Matrix Algebra on GPU and Multicore Architectures)

• The MAGMA team has made the conclusion that dense linear algebra methods are now a better fit on GPU architectures instead of traditional multicore architectures
  – (Nath, Stanimire, & Dongarra, 2010)

• MAGMA BLAS libraries up to **838 Gflop/s**
  – 33% occupancy and 2 thread blocks per SM (Volkov, 2010)
Strategies for embracing heterogeneous computing.

– Opportunities enabled by CUDA x86
– Practical ideas for balancing CPU & GPU
– Practical tips on running CUDA Kernels on CPU cores

A brand new area where everyone is learning

Do I foresee this as an important topic in the future?

Yes!
Items of note (slide 1)

• The **size of a warp will be different** from the expected 32 threads per warp for a GPU.
  – For x86 computing a warp might be the size of the SIMD units on the x86 core (either four or eight) or one thread per warp when SIMD execution is not utilized

• **Synchronization is different**: The compiler will remove explicit synchronization of the thread processors when it can determine that it is safe to split loops where the synchronization calls occur
Items of note (slide 2)

- **Still have explicit movement** of data between host and device memory and global to shared memory
  - The PGI compiler allows pointer swapping on x86 systems.
  - Perhaps a wrapper around cudaMemcpy()?
- **Watch out for PCIe configuration!**
  - Especially for benchmarks that hide poor configurations
Find a mapping that reuses data $\text{energy} = \text{objFunc}(p_1, p_2, \ldots p_n)$

Optimization Method
(Powell, Conjugate Gradient, Other)

- **Step 1** Broadcast parameters
- **Step 2** Calculate partials
- **Step 3** Sum partials to get energy

**GPU 1**
$p_1, p_2, \ldots p_n$
Examples
0, N-1

**GPU 2**
$p_1, p_2, \ldots p_n$
Examples
N, 2N-1

**GPU 3**
$p_1, p_2, \ldots p_n$
Examples
2N, 3N-1

**GPU 4**
$p_1, p_2, \ldots p_n$
Examples
3N, 4N-1
## Speedup over a quad core

<table>
<thead>
<tr>
<th>OS</th>
<th>Machine</th>
<th>Opt method</th>
<th>Precision</th>
<th>Ave obj func time</th>
<th>% func time</th>
<th>Speedup over quad-core</th>
<th>Speedup over single-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.00532</td>
<td>100.0</td>
<td>85</td>
<td>341</td>
</tr>
<tr>
<td>Win7</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.00566</td>
<td>100.0</td>
<td>81</td>
<td>323</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA GTX280</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.01109</td>
<td>99.2</td>
<td>41</td>
<td>163</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>64</td>
<td>0.01364</td>
<td>100.0</td>
<td>40</td>
<td>158</td>
</tr>
<tr>
<td>Win7</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>64</td>
<td>0.01612</td>
<td>100.0</td>
<td>22</td>
<td>87</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Levenberg-Marquardt</td>
<td>32</td>
<td>0.04313</td>
<td>2.7</td>
<td>10</td>
<td>38</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Levenberg-Marquardt</td>
<td>64</td>
<td>0.08480</td>
<td>4.4</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td>Levenberg-Marquardt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```c
#pragma omp parallel for reduction(+ : sum)
for(int i=0; i < nExamples; ++i) {
    Real d = getError(i);
    sum += d;
}
```
The CUDA execution model

• Loose coupling between SM translates to strong scaling (even on CPU cores) – very good news!
• On x86: beware SMP scaling limits caused by cache coherency (AMD Barcelona example on TACC Ranger)

```c
#pragma omp parallel for reduction(+ : sum)
    for(int i=0; i < nExamples; ++i) {
        Real d = getError(i);
        sum += d;
    }
```

Likely cause: some AMD cache coherency messages take two hops
Task parallelism

- Asynchronous kernel launches will become more important (task vs. data parallelism)
  - x86 great for task parallelism
- Interesting to see how prevalent use will affect CUDA
  - Reduction to a single value does not naturally fit in the CUDA model as it requires:
    - Atomic operations (scalability issues!)
    - Separate kernels (rule 2: startup overhead)
    - Transfer to the host for the final step
Map irregular data structures to the CPU

A gather operation
for(int i=0; i < n; i++)
a[i] = b[index[i]]

The GPU L2 cache cannot help with large data

<table>
<thead>
<tr>
<th>Size</th>
<th>Op</th>
<th>nTests</th>
<th>Time</th>
<th>Slowdown relative to sequential performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01M</td>
<td>Sequential</td>
<td>1000</td>
<td>3.37E-06</td>
<td></td>
</tr>
<tr>
<td>0.01M</td>
<td>Sorted</td>
<td>1000</td>
<td>3.44E-06</td>
<td>1.0</td>
</tr>
<tr>
<td>0.01M</td>
<td>Random</td>
<td>1000</td>
<td>7.46E-06</td>
<td>2.2</td>
</tr>
<tr>
<td>0.1M</td>
<td>Sequential</td>
<td>1000</td>
<td>1.39E-05</td>
<td>1.0</td>
</tr>
<tr>
<td>0.1M</td>
<td>Sorted</td>
<td>1000</td>
<td>1.42E-05</td>
<td>1.0</td>
</tr>
<tr>
<td>0.1M</td>
<td>Random</td>
<td>1000</td>
<td>6.94E-05</td>
<td>5.0</td>
</tr>
<tr>
<td>1M</td>
<td>Sequential</td>
<td>1000</td>
<td>0.000107</td>
<td></td>
</tr>
<tr>
<td>1M</td>
<td>Sorted</td>
<td>1000</td>
<td>0.000106</td>
<td>1.0</td>
</tr>
<tr>
<td>1M</td>
<td>Random</td>
<td>1000</td>
<td>0.000972</td>
<td>9.1</td>
</tr>
<tr>
<td>10M</td>
<td>Sequential</td>
<td>1000</td>
<td>0.001077</td>
<td></td>
</tr>
<tr>
<td>10M</td>
<td>Sorted</td>
<td>1000</td>
<td>0.00105</td>
<td>1.0</td>
</tr>
<tr>
<td>10M</td>
<td>Random</td>
<td>1000</td>
<td>0.011418</td>
<td>10.6</td>
</tr>
<tr>
<td>100M</td>
<td>Sequential</td>
<td>1000</td>
<td>0.011553</td>
<td></td>
</tr>
<tr>
<td>100M</td>
<td>Sorted</td>
<td>1000</td>
<td>0.013233</td>
<td>1.1</td>
</tr>
<tr>
<td>100M</td>
<td>Random</td>
<td>1000</td>
<td>0.132465</td>
<td>11.5</td>
</tr>
</tbody>
</table>

GPU Computing Gems is an excellent resource
There is certainly much, much more

Thank you!
CUDA Application Design and Development is now available for preorder

http://www.amazon.com/CUDA-Application-Design-Development-Farber/dp/0123884268
Acknowledgements

Supported by Science Foundation Ireland under grant 08/HEC/I1450 and by HEA’s PRTLI-C4.