Enabling Science through Emerging HPC Technologies

N.S. Scott, I. Spence and C.J. Gillan

School of Electronics, Electrical Engineering & Computer Science
Queen’s University Belfast

7th July 2009
5 free programs for each conference participant

To choose your programs:-

1. Access the web site
http://cpc.cs.qub.ac.uk

2. Click on the Subject Index link on the toolbar

3. Click on the category in which you are interested.

4. Click on the Catalogue Identifier for a brief description of the program.

5. E-mail your 5 choices to cpc@qub.ac.uk
The throes of a revolution?
The throes of a revolution?

Emerging technologies

1. NVIDIA GPU
2. STI Cell processor
3. FPGA
4. Intel Larrabee
5. IBM Roadrunner

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core2 Quad Extreme</td>
<td>192GF</td>
<td>96GF</td>
<td>12.6GB/s</td>
</tr>
<tr>
<td>NVIDIA C1060</td>
<td>933GF</td>
<td>78GF</td>
<td>102GB/s</td>
</tr>
</tbody>
</table>
Overview

1. Only single precision arithmetic is available
2. The parallelism has to be SIMD (Single Instruction, Multiple Data)
3. The amount of on-chip memory available to each thread is limited
4. Use of texture memory is important
The problem

To evaluate millions of integrals of the form

\[ \int_0^a f_i(r) \, dr \]
The problem

To evaluate millions of integrals of the form

\[
\int_0^a f_i(r) \left[ \int_0^r \left( \frac{t}{r} \right)^\lambda g_i(t) dt + \int_r^a \left( \frac{r}{t} \right)^{\lambda+1} g_i(t) dt \right] dr
\]
The problem

To evaluate millions of integrals of the form

\[ \int_0^a f_i(r) \left[ \int_0^r \left( \frac{t}{r} \right)^\lambda g_i(t) \, dt + \int_r^a \left( \frac{r}{t} \right)^{\lambda+1} g_i(t) \, dt \right] \, dr \]

The pattern of computation can be illustrated by the following incomplete code (e.g. \( \left( \frac{t}{r} \right)^\lambda \) terms are omitted).

```c
for (r=2; r < numOfSteps-1; r+=2)
{
    inner[r] = inner[r-2] + g(r-2) + 4*g(r-1) + g(r);
    inner[r+1] = inner[r-1] + g(r-1) + 4*g(r) + g(r+1);
}
... outer += 4*f(r)*inner[r] + 2*f(r+1)*inner[r+1];
```
1. Used in calculations of electron scattering by the hydrogen atom.

2. Simpson’s rule (with the same mesh of \( \approx 1000 \) elements) used for evaluation of the inner and outer integrals.

3. The functions \( f_i(r) \) and \( g_i(r) \) (orbitals) can be pre-calculated and fit in global memory but the access patterns of the different integrals are complex.

4. Each integral involves a local temporary array of \( 2 \times 1000 \) elements.
Under/overflow

The original algorithm pre-computed all possible values of $r^\lambda$ and $\frac{1}{r^\lambda}$.
This resulted in actual underflow and potential overflow when moved to single precision.
The solution was to recast the algorithm to use ratios of the form $(r_{n-1}/r_n)$. 

Accuracy

The single precision program agrees with the original double precision version within a relative error of 0.001% or an absolute error of $10^{-7}$ (whichever is greater).
Single precision arithmetic

Under/overflow
The original algorithm pre-computed all possible values of $r^\lambda$ and $\frac{1}{r^\lambda}$.
This resulted in actual underflow and potential overflow when moved to single precision.
The solution was to recast the algorithm to use ratios of the form $(r_{n-1}/r_n)$.

Accuracy
The single precision program agrees with the original double precision version within a relative error of 0.001% or an absolute error of $10^{-7}$ (whichever is greater).
On-chip memory is limited

**Original approach**

For each outer integral, the values of approximately $2 \times 1000$ inner integrals are required and the original algorithm stored these in arrays. This (replicated for each thread) was far too much to be stored in on-chip GPU memory.
On-chip memory is limited

Original approach
For each outer integral, the values of approximately $2 \times 1000$ inner integrals are required and the original algorithm stored these in arrays. This (replicated for each thread) was far too much to be stored in on-chip GPU memory.

Revised approach
Evaluating the outer integral using Simpson’s rule is iterative and it proved possible to calculate an inner value, add its contribution to the outer integral and then throw it away. Storage of all the values was found to be unnecessary.
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages
1. Simple
2. Distributes load well if threads are asynchronous

Disadvantage
1. Not SIMD - on the GPU the threads become sequential on each multiprocessor
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages
1. Simple
2. Distributes load well if threads are asynchronous

Disadvantage
1. Not SIMD - on the GPU the threads become sequential on each multiprocessor
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages
1. Simple
2. Distributes load well if threads are asynchronous

Disadvantage
1. Not SIMD - on the GPU the threads become sequential on each multiprocessor

N.S. Scott, I. Spence and C.J. Gillan (QUB) Enabling Science... HPC Technologies
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages

1. Simple
2. Distributes load well if threads are asynchronous

Disadvantage

1. Not SIMD - on the GPU the threads become sequential on each multiprocessor
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages

1. Simple
2. Distributes load well if threads are asynchronous

Disadvantage

1. Not SIMD - on the GPU the threads become sequential on each multiprocessor
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i
SIMD Parallelism was an issue

Original threading structure

Problem instances – indexed by i

if (i % nThreads == threadId)

Evaluate integral i

Advantages

1. Simple
2. Distributes load well if threads are asynchronous
SIMD Parallelism was an issue

**Original threading structure**

```
Problem instances – indexed by i
```

```
if (i % nThreads == threadId)
```

```
Evaluate integral i
```

**Advantages**

1. Simple
2. Distributes load well if threads are asynchronous

**Disadvantage**

1. Not SIMD - on the GPU the threads become sequential on each multiprocessor
Each integral is evaluated by loops with fixed bounds. As long as each GPU thread is just evaluating an integral, SIMD operation is achieved.

```c
for (i=1; i < numOfSteps-1; i++)
{
    nextInner = prevInner + ... 
    nextOuter = prevOuter + ... nextInner ... 
    prevInner = nextInner; prevOuter = nextOuter;
}
```
Have no conditionals on GPU

Each integral is evaluated by loops with fixed bounds. As long as each GPU thread is just evaluating an integral, SIMD operation is achieved.

```c
for (i=1; i< numOfSteps-1; i++)
{
    nextInner = prevInner + ...
    nextOuter = prevOuter + ... nextInner ...
    prevInner = nextInner; prevOuter = nextOuter;
}
```

On the host, the values required to specify integrals are accumulated into batches. When enough have been collected, they are passed to the GPU to be evaluated concurrently by multiple threads. The results are then returned to the host.
GPU threads do the same thing at the same time

Host

Orbitals

Parameters for integrals

Results

GPU

Orbitals

Parameters for integrals

GPU Threads

Results
GPU threads do the same thing at the same time.

![Diagram showing the relationship between Host and GPU with Orbitals, Parameters for integrals, and Results.]
GPU threads do the same thing at the same time
GPU threads do the same thing at the same time.
GPU threads do the same thing at the same time.
GPU threads do the same thing at the same time
GPU threads do the same thing at the same time
GPU threads do the same thing at the same time
Using texture memory

Appropriate access pattern
The use made of the functions $f_i$ and $g_i$ (the orbitals) is appropriate for using *texture memory* on the device.

1. The values are not modified by the kernel functions
2. There is some locality of access

Benefits
The caching of texture memory makes its access considerably faster than global memory.
Use lots of GPU threads

Elapsed time for varying number of GPU threads (8 million integrals)

- Linear speedup
- Using GPU

Elapsed time (seconds)

Number of GPU threads (on each GPU)
Times for $10^{12}$ floating point operations

Elapsed time for varying numbers of host cores and GPUs (43 million integrals)

Number of cores

Wall clock time (seconds)

Number of GPUs

2 GPUs, 26.9 seconds

4 cores, 435 seconds
Conclusions: Myth or Reality

### Myth

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA C1060</td>
<td>933GF</td>
<td>78GF</td>
<td>102GB/s</td>
</tr>
</tbody>
</table>

Considerable performance benefits may well be available if single precision is good enough and your algorithm can be expressed in terms of SIMD parallelism.

- Use a large number (thousands) of GPU threads.
- To evaluate 43 million integrals ($\approx 10^{12} = 1000$GFlop) took 435 seconds on 4 cores, 27 seconds on 2 GPUs ($\approx 30$GFlop/s).
Conclusions: Myth or Reality

### Myth

<table>
<thead>
<tr>
<th></th>
<th>Single</th>
<th>Double</th>
<th>Bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA C1060</td>
<td>933GF</td>
<td>78GF</td>
<td>102GB/s</td>
</tr>
</tbody>
</table>

### Reality

1. Considerable performance benefits may well be available if single precision is good enough and your algorithm can be expressed in terms of SIMD parallelism.
2. Use a large number (thousands) of GPU threads.
3. To evaluate 43 million integrals ($\approx 10^{12} = 1000$GFlop) took 435 seconds on 4 cores, 27 seconds on 2 GPUs ($\approx 30$GFlop/s 😞).
“When vector processing declined in favour of computational clusters, SIMD processing appeared to fade from the mainstream. It could be argued that SIMD processing is now undergoing a rebirth as it is becoming a centrepiece in modern architectures such as the Cell BE, modern general purpose graphics processors and likely the fruits of the Intel Larrabee project as well.”

\[ \langle ij|kl \rangle = \int dx_1 dx_2 \chi_i^*(x_1)\chi_j^*(x_2) \frac{1}{r_{12}} \chi_k(x_1)\chi_l(x_2). \] (2)

Massive thread-level parallelism inherent on each SCF iteration

---

“On modern architectures, the performance of 32-bit operations is often at least twice as fast as the performance of 64-bit operations. By using a combination of 32-bit and 64-bit floating point arithmetic performance can be significantly enhanced while maintaining the 64-bit accuracy of the resulting solution.”

Iterative refinement for dense systems, $Ax = b$, can work this way.

---

$\text{L U} = \text{lu}(A)$
$x = L \backslash (U \backslash b)$
$r = b - Ax$

\begin{itemize}
  \item WHILE $\| r \|$ not small enough
    \begin{itemize}
      \item $z = L \backslash (U \backslash r)$
      \item $x = x + z$
      \item $r = b - Ax$
    \end{itemize}
  \end{itemize}

---

\footnote{\textit{“Accelerating scientific computations with mixed precision algorithms,”} Marc Baboulin, Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julie Langou, Julien Langou, Piotr Luszczek, Stanimire Tomov, CPC (2009), doi:10.1016/j.cpc.2008.11.005}
SP vs DP fp performance

1. Requires extra storage, total is 1.5 times normal
2. $O(n^3)$ work is done in lower precision
3. $O(n^2)$ work is done in high precision
“Novel data formats like Block Data Layout (BDL) improve locality of reference to memory and higher reuse of data in memories that are closer to cores (caches or scratchpad memories).”

http://icl.cs.utk.edu/plasma
Programming is difficult

QP: A Heterogeneous Multi-Accelerator Cluster

Programming is difficult

Table 2. Applications developed to run on the multi-core/GPU/FPGA cluster.

<table>
<thead>
<tr>
<th>Application</th>
<th># of CPU cores used</th>
<th># of GPUs used</th>
<th># of FPGAs used</th>
<th>non-accelerated application performance</th>
<th>accelerated application performance</th>
<th>overall speedup achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMD (molecular dynamics) [20]</td>
<td>60</td>
<td>60</td>
<td></td>
<td>0.471 sec/step</td>
<td>0.085 sec/step</td>
<td>5.5X</td>
</tr>
<tr>
<td>WRF (weather prediction) [21]</td>
<td>48</td>
<td>48</td>
<td></td>
<td>79.9 sec</td>
<td>64.9 sec</td>
<td>1.23X</td>
</tr>
<tr>
<td>ACF (cosmology) [22]</td>
<td>8</td>
<td></td>
<td>8</td>
<td>5,537.7 sec</td>
<td>880.8 sec</td>
<td>6.3X</td>
</tr>
<tr>
<td>ACF (cosmology) [22]</td>
<td>32</td>
<td>32</td>
<td></td>
<td>1,476.8 sec</td>
<td>70.1 sec</td>
<td>21.1X</td>
</tr>
</tbody>
</table>
Programming is difficult

### Table 2. Applications developed to run on the multi-core/GPU/FPGA cluster.

<table>
<thead>
<tr>
<th>Application</th>
<th># of CPU cores used</th>
<th># of GPUs used</th>
<th># of FPGAs used</th>
<th>non-accelerated application performance</th>
<th>accelerated application performance</th>
<th>overall speedup achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAMD (molecular dynamics) [20]</td>
<td>60</td>
<td>60</td>
<td></td>
<td>0.471 sec/step</td>
<td>0.085 sec/step</td>
<td>5.5X</td>
</tr>
<tr>
<td>WRF (weather prediction) [21]</td>
<td>48</td>
<td>48</td>
<td></td>
<td>79.9 sec</td>
<td>64.9 sec</td>
<td>1.23X</td>
</tr>
<tr>
<td>ACF (cosmology) [22]</td>
<td>8</td>
<td>8</td>
<td></td>
<td>5,537.7 sec</td>
<td>880.8 sec</td>
<td>6.3X</td>
</tr>
<tr>
<td>ACF (cosmology) [22]</td>
<td>32</td>
<td>32</td>
<td></td>
<td>1,476.8 sec</td>
<td>70.1 sec</td>
<td>21.1X</td>
</tr>
</tbody>
</table>

Tower of Babel “.. Come let us go down and confuse their language so that they will not understand each other ..”, Genesis 11
Programming is difficult

C++ generic template programming

HONEI: A collection of libraries for numerical computations targeting multiple processor architectures.\textsuperscript{a}

Challenging times ahead for HPC

1. Algorithms that can exploit the memory hierarchy
2. Mixed precision algorithms
3. Performance portability
4. Adaptive and self tuning algorithms
5. New ways of programming
Challenging times ahead for HPC

1. Algorithms that can exploit the memory hierarchy
2. Mixed precision algorithms
3. Performance portability
4. Adaptive and self-tuning algorithms
5. New ways of programming

Later Today - Section 5 14:30-16:00

176
E. Perepelkin, V. Smirnov
Beam Dynamic Calculation by NVIDIA CUDA Technology

179
A.V. Adinetz
C$: A High-Level Programming System for High-Performance Graphics Processors