Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels

Joost Huizinga

February 17, 2012
Multi-Core platforms: the current solution to battle the physical limitations of microelectronics.
- Examples: Multi-core CPU, STI Cell/B.E., GPU

Multi-Core platforms are the main platforms for High Performance Computing (HPC) applications.

Many HPC applications are computational intensive.

We will focus on data-intensive applications, which have different requirements.

So, for data-intensive applications, which Multi-Core platform is best?
Outline

- The Problem
- The Platforms
- The Experiment
- The Results
- The Conclusion
The Problem

- Our problem comes from radio astronomy.
- Dozens of antenna stations generate data in the order of Tbit/s.
- Data must be processed on-line to reduce data storage.
- A data-intensive operation of this process is called *convolutional resampling*.
  - Converts an incoming data stream into a grid model of the sky.
- Our problem is a small part of this process called *gridding*.
The problem: Gridding

- An antenna generates a stream of **samples**.
- Each sample contains about a thousand different **frequencies**.
- Every sample-frequency combination has a data element called a **visibility**.
- Every visibility is multiplied with a row in the **convolution matrix**.
- The resulting array is added, as a block, to the output matrix called the **grid**.
- The row in the convolution matrix and the block in the grid can be different for every visibility.
The problem: Gridding pseudo code

```c
for each (sample, freq) {
    cIndex = cOffset(sample.coordinate, freq);
    gIndex = gOffset(sample.coordinate, freq);

    for (x=0; x<M; x++) {
        grid[gIndex + x] += C[cIndex + x] * sample.vis(freq);
    }
}
```

Pseudo code for the gridding computation.

- **sample**: A sample containing a coordinate (coordinate) and one visibility (vis) for each frequency (freq).
- **freq**: A frequency.
- **M**: The convolution kernel size.
- **C**: The convolution matrix.
- **grid**: The output grid.
The problem: Gridding pseudo code

```c
for (sample, freq) {
    cIndex = cOffset(sample.coordinate, freq);
    gIndex = gOffset(sample.coordinate, freq);

    for (x=0; x<M; x++){
        grid[gIndex + x] += C[cIndex + x] * sample.vis(freq);
    }
}
```

Pseudo code for the gridding computation.

- **Important property 1:**
  - The offset functions (*line 2 and 3*) return unpredictable and unbalanced offsets.
  - Splitting data on samples means that cores need to retrieve a lot of data from memory.
  - Splitting data on the C matrix causes load imbalance.
The problem: Gridding pseudo code

```plaintext
foreach(sample, freq){
    cIndex = cOffset(sample.coordinate, freq);
    gIndex = gOffset(sample.coordinate, freq);

    for (x=0; x<M; x++){
        grid[gIndex + x] += C[cIndex + x] * sample.vis(freq);
    }
}
```

Pseudo code for the gridding computation.

- **Important property 2:**
  The indexes modified by `x` (line 5), are predictable.
  - Increasing the kernel size `M` will thus reduce data access irregularity.
The problem: Gridding pseudo code

```plaintext
foreach (sample, freq) {
    cIndex = cOffset(sample.coordinate, freq);
    gIndex = gOffset(sample.coordinate, freq);

    for (x=0; x< M; x++){
        grid[gIndex + x] += C[cIndex + x] * sample.vis(freq);
    }
}
```

Pseudo code for the gridding computation.

**Important property 3:**
Line 6 has a low arithmetic intensity and is the most executed line in the code.
- This line will be executed $N_{\text{sample}} \times N_{\text{freq}} \times M$ times.
- This line contains 3 memory operations (2 reads, 1 write) and 1 floating point operation (*multiplication*), which gives an arithmetic intensity of 0.33.
Three Multi-Core platforms were evaluated:
- Multi-core CPU (*MC-CPU*)
- STI Cell Broadband Engine (*Cell/B.E.*)
- GPU
The Platforms: Multi-Core CPU

- **Memory model:** Shared memory, Coherent caches
- **Core type:** Homogeneous Cores
- **Programming model:** Multi-Threading
- **Task scheduling:** By operating system
The Platforms: Multi-Core CPU Programming Specifics

- Each thread works on a private grid, merged in a final reduction step.
- Task distribution is done according to a master-worker approach.
- The master optimizes data locality by giving overlapping tasks to the same worker.
- In-core performance of workers is improved with SIMD. *(SIMD: Single Instruction, Multiple Data)*
## The Platforms: Multi-Core CPU

### Table: Multi-Core CPU Platform Instances

<table>
<thead>
<tr>
<th>Platform Instance</th>
<th>Cores</th>
<th>Clock (GHz)</th>
<th>Memory per Core (kB)</th>
<th>Main Memory</th>
<th>Compute Rate (GFlop/s)</th>
<th>Memory Bw (GB/s)</th>
<th>Flop/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Intel Xeon 5320</td>
<td>2x4</td>
<td>1.86</td>
<td>32+32 L1 2x4096 L2</td>
<td>8 GB</td>
<td>59.5</td>
<td>21.3</td>
<td>2.8</td>
</tr>
<tr>
<td>Quad AMD Opteron 8212</td>
<td>4x2</td>
<td>2.00</td>
<td>64+64 L1 2x1024 L2</td>
<td>4 GB</td>
<td>64.0</td>
<td>10.7</td>
<td>6.0</td>
</tr>
<tr>
<td>Intel Core i7 920</td>
<td>4 (HT)</td>
<td>2.66</td>
<td>32+32 L1 256 L2 1x8192 L3</td>
<td>6 GB</td>
<td>85.2</td>
<td>32.0</td>
<td>2.7</td>
</tr>
</tbody>
</table>
- **Memory model**: Distributed memory
- **Core type**: Heterogeneous cores: One PPE, Multiple SPEs  
  *PPE: Power Processing Element; SPE: Synergistic Processing Element*
- **Programming model**: Cell SDK
- **Task scheduling**: By application
The Memory Flow Controller (MFC) is used to handle overlapping write operations in the grid.

Task distribution is done according to a master-worker approach.

The master optimizes data locality by giving overlapping tasks to the same worker.

Basic SPE-specific optimizations have been applied.
## The Platforms: STI Cell Broadband Engine

### Table: STI Cell/B.E. platform instances

<table>
<thead>
<tr>
<th>Platform Instance</th>
<th>Cores</th>
<th>Clock (GHz)</th>
<th>Memory per Core (kB)</th>
<th>Main Memory</th>
<th>Compute Rate (GFlop/s)</th>
<th>Memory Bw (GB/s)</th>
<th>Flop/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>STI PS3-Cell/B.E.</td>
<td>1+6</td>
<td>3.20</td>
<td>256</td>
<td>256 MB</td>
<td>153.6</td>
<td>25.6</td>
<td>6.0</td>
</tr>
<tr>
<td>STI Cell/B.E.</td>
<td>1+8</td>
<td>3.20</td>
<td>256</td>
<td>1 GB</td>
<td>204.8</td>
<td>25.6</td>
<td>8.0</td>
</tr>
<tr>
<td>STI QS21-Cell/B.E.</td>
<td>2+16</td>
<td>3.20</td>
<td>256</td>
<td>2 GB</td>
<td>409.6</td>
<td>51.2</td>
<td>8.0</td>
</tr>
</tbody>
</table>

---

**STI CELL Processor**

---

[Joost Huizinga](#)  
Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels
- **Memory model**: Distributed memory, Small Chaches
- **Core type**: Homogeneous Cores
- **Programming model**: CUDA, SIMT  
  *(CUDA: Compute Unified Device Architecture; SIMT: Single Instruction, Multiple Threads)*
- **Task scheduling**: By hardware
The Platforms: GPU

- Each block of threads writes to a private grid.
- Task distribution is done by block partitioning of visibility data.
- Access to the convolution matrix is optimized with texture fetching.
- Writes to the grid are made aligned and contiguous to create coalesced memory access and improve memory bandwidth.
- No data-dependent optimizations or job distributions have been applied.

Joost Huizinga
Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels
### Table: GPU platform instances

<table>
<thead>
<tr>
<th>Platform Instance</th>
<th>Cores</th>
<th>Clock (GHz)</th>
<th>Memory per Core (kB)</th>
<th>Main Memory</th>
<th>Compute Rate (GFlop/s)</th>
<th>Memory Bw (GB/s)</th>
<th>Flop/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>NVIDIA GeForce 8800 GTX</td>
<td>16</td>
<td>1.35</td>
<td>16+8+8</td>
<td>768 MB</td>
<td>345.6</td>
<td>86.4</td>
<td>4.0</td>
</tr>
<tr>
<td>NVIDIA Tesla C1060</td>
<td>30</td>
<td>1.30</td>
<td>16+8+8</td>
<td>4 GB</td>
<td>936.0</td>
<td>102.0</td>
<td>9.2</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 280</td>
<td>30</td>
<td>1.30</td>
<td>16+8+8</td>
<td>1 GB</td>
<td>936.0</td>
<td>141.7</td>
<td>6.6</td>
</tr>
</tbody>
</table>

---

The Platforms: GPU

Joost Huizinga

Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels
# The Platforms: Overview

## Table: Platform Overview

<table>
<thead>
<tr>
<th>Platform</th>
<th>Memory Model</th>
<th>Core Type</th>
<th>Programming Model</th>
<th>In-Core Optimizations</th>
<th>Data Distribution</th>
<th>Task Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC-CPU</td>
<td>Shared memory, Coherent caches</td>
<td>Homogeneous</td>
<td>Multi-Threading</td>
<td>SIMD</td>
<td>Master-Worker Queues</td>
<td>OS</td>
</tr>
<tr>
<td>Cell</td>
<td>Distributed memory</td>
<td>Heterogeneous</td>
<td>Cell SDK</td>
<td>SIMD, Multi-Buffering</td>
<td>Master-Worker Queues</td>
<td>Application</td>
</tr>
<tr>
<td>GPU</td>
<td>Distributed memory, Small caches</td>
<td>Homogeneous</td>
<td>CUDA SIMT</td>
<td></td>
<td>Symmetric Blocks</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

## Table: Platform Instances Overview

<table>
<thead>
<tr>
<th>Platform Instance</th>
<th>Cores</th>
<th>Clock (GHz)</th>
<th>Memory per Core (kB)</th>
<th>Main Memory</th>
<th>Compute Rate (GFlop/s)</th>
<th>Rate</th>
<th>Memory Bw (GB/s)</th>
<th>Flop/Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Intel Xeon 5320</td>
<td>2x4</td>
<td>1.86</td>
<td>32+32 L1 2x4096 L2</td>
<td>8 GB</td>
<td>59.5</td>
<td>21.3</td>
<td>2.8</td>
<td></td>
</tr>
<tr>
<td>Quad AMD Opteron 8212</td>
<td>4x2</td>
<td>2.00</td>
<td>64+64 L1 2x1024 L2</td>
<td>4 GB</td>
<td>64.0</td>
<td>10.7</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>Intel Core i7 920</td>
<td>4</td>
<td>2.66 (HT)</td>
<td>32+32 L1 256 L2 1x8192 L3</td>
<td>6 GB</td>
<td>85.2</td>
<td>32.0</td>
<td>2.7</td>
<td></td>
</tr>
<tr>
<td>STI PS3-Cell/B.E.</td>
<td>1+6</td>
<td>3.20</td>
<td>256</td>
<td>256 MB</td>
<td>153.6</td>
<td>25.6</td>
<td>6.0</td>
<td></td>
</tr>
<tr>
<td>STI QS21-Cell/B.E.</td>
<td>1+8</td>
<td>3.20</td>
<td>256</td>
<td>1 GB</td>
<td>204.8</td>
<td>25.6</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>STI 2xQS21-Cell/B.E.</td>
<td>2+16</td>
<td>3.20</td>
<td>256</td>
<td>2 GB</td>
<td>409.6</td>
<td>51.2</td>
<td>8.0</td>
<td></td>
</tr>
<tr>
<td>NVIDIA GeForce 8800 GTX</td>
<td>16</td>
<td>1.35</td>
<td>16+8+8 768 MB</td>
<td>86.4</td>
<td>4.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA Tesla C1060</td>
<td>30</td>
<td>1.30</td>
<td>16+8+8 4 GB</td>
<td>102.0</td>
<td>9.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 280</td>
<td>30</td>
<td>1.30</td>
<td>16+8+8 1 GB</td>
<td>141.7</td>
<td>6.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Experiment

- Each platform instance had to run several different instances of the gridding computation.
- These variables were different for each gridding instance:
  - Kernel size
  - Arithmetic intensity factor
  - Applied optimizations
- The experiment was not fully factorial: every variable has been investigated on its own.
- Performance was measured in GFlop/s.
The Experiment: Kernel size

- Basically the width of the convolution matrix.
- Affects the data access irregularity: the smaller the kernel size, the higher the irregularity.
- Kernel sizes tested: 16x16, 32x32, 64x64, 128x128.
The Experiment: Arithmetic intensity factor

- The factor with which the arithmetic intensity has increased compared to the base value of 0.33.
- The factor was increased by adding floating point operations to the ‘data-intensive’ line of the code.
- Arithmetic intensity factors tested: 1x, 2x, 3x, 6x, 12x, 24.
The gridding kernel was evaluated at various points during the optimization process.

For every platform there is a base, without any optimization, and three optimization steps.

For every optimization it was noted how many days it took to apply the optimization.

The optimization steps are different for every platform.
• **Base:** The sequential algorithm.

• **Step 1:** Simple parallelization with symmetrical data distribution.

• **Step 2:** Implementation of the Master-Worker system.

• **Step 3:** Worker queues are sorted locally (by the worker).
The Experiment: Applied optimizations for the Cell/B.E.

- **Base:** The sequential algorithm with symmetrical data distribution.
- **Step 1:** Replaced symmetrical data distribution by a dynamic one.
- **Step 2:** The master gives overlapping jobs to the same worker.
- **Step 3:** Worker queues are sorted locally (by the worker).
The Experiment: Applied optimizations for the GPU

- **Base**: The sequential algorithm.
- **Step 1**: Use of modified CUBLAS routines.
- **Step 2**: Use of coalesced memory access.
- **Step 3**: Minor optimizations, such as using the texture caches.
The Results: Data Access Irregularity

Result vary, though differences are relatively small.

No platform comes close to its actual capacity.

Application performance (GFlops) running with various kernel sizes.

Joost Huizinga
Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels
The Results: Data Access Irregularity

- Result vary, though differences are relatively small.
- No platform comes close to its actual capacity.

Application performance (GFlops) running with various kernel sizes.
The Results: Data Access Irregularity

- For small kernels the fastest platform instance is the MC-CPU i7 Core 920, 16th.
- For large kernels the fastest platform instance is the 2x Cell in QS21.

Application performance (GFlops) running with various kernel sizes.
The Results: Data Access Irregularity

- In general: it depends.

Application performance (GFlops) running with various kernel sizes.
The Results: Arithmetic Intensity

Differences become larger as Arithmetic intensity increases. The GPUs and Cell/B.E. start to climb to their maximum capacity.

Figure: i7 Core 920
Figure: 2x Cell in QS21
Figure: All different GPUs

Application Performance (GFlop/s) running with various arithmetic intensities.

Joost Huizinga
Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels
The Results: Arithmetic Intensity

- Differences become larger as Arithmetic intensity increases
- The GPUs and Cell/B.E. actual start to climb to their maximum capacity.

**Figure: i7 Core 920**

**Figure: 2x Cell in QS21**

**Figure: All different GPUs**

Application Performance (GFlop/s) running with various arithmetic intensities.
For low arithmetic intensities the GPU GTX 280 seems to be fastest.

For high arithmetic intensities the GPU C1060 is fastest.

Application Performance (GFlop/s) running with various arithmetic intensities.
The Results: Arithmetic Intensity

- In general: GPUs are the best choice for high arithmetic intensities.

Figure: i7 Core 920

Figure: 2x Cell in QS21

Figure: All different GPUs

Application Performance (GFlop/s) running with various arithmetic intensities.
The optimisations are wildly different and difficult to compare.

*Days needed to implement* is a rather unstable measure of effort.

- Programming effort might change drastically if programmers gain more experience.
- The Cell/B.E. gained most from the applied optimizations.
  - However: There might be other optimizations that change this in favor of (one of) the other two platforms.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MC-CPU:</td>
<td>Base</td>
<td>1.6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dual Xeon 5320</td>
<td>1</td>
<td>7</td>
<td>4.4x</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>18</td>
<td>11.3x</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24.8</td>
<td>15.5x</td>
<td>4</td>
</tr>
<tr>
<td>Cell/B.E.:</td>
<td>Base</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2x STI QS21</td>
<td>1</td>
<td>6</td>
<td>3.0x</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>7.0x</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>57</td>
<td>23.5x</td>
<td>3</td>
</tr>
<tr>
<td>GPU:</td>
<td>Base</td>
<td>1.6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>GeForce 8800 GTX</td>
<td>1</td>
<td>4.9</td>
<td>3.1x</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>22.3</td>
<td>13.9x</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23.8</td>
<td>14.9x</td>
<td>12</td>
</tr>
</tbody>
</table>

*Table*: Optimization gain and effort for different platforms
The Results: Overview

- The table below assumes low arithmetic intensity and large kernel size.
- Even then it is difficult to pick a ‘best system’.
- Also, the properties in the table below might change (rapidly) over time.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Perf.</th>
<th>Scal.</th>
<th>Cost</th>
<th>Effort</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC-CP</td>
<td>~</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td>~</td>
</tr>
<tr>
<td>Cell/B.E.</td>
<td>++</td>
<td>++</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>GPU</td>
<td>+</td>
<td>-</td>
<td>~</td>
<td>~</td>
<td>~</td>
</tr>
</tbody>
</table>

Table: An overview of the tested platforms
The Platform can have a lot of influence on the overall performance.
However: ‘best overall platform’ does not exist.
So, if performance, cost, scalability and time are all important issues in your next HPC project:
The Platform can have a lot of influence on the overall performance.

However: ‘best overall platform’ does not exist.

So, if performance, cost, scalability and time are all important issues in your next HPC project:

Choose your platform carefully!
The Platform can have a lot of influence on the overall performance.

However: ‘best overall platform’ does not exist.

So, if performance, cost, scalability and time are all important issues in your next HPC project:

Choose your platform carefully!

That is it, any questions?