
Rengan Xu, Sunita Chandrasekaran, Barbara Chapman, Christoph F. Eick

WOLFHPC’12 in conjunction with SC12

November 16, 2012
Outline

- Motivation
- Related Work
- Overview of Directive-based Models
- Experiments and Results
- Conclusion and Future Work
Motivation

• GPUs have high compute capability in HPC, but programming these devices is a challenge

• Low-level models: CUDA, OpenCL
  • Language extension
  • Time-consuming to write and error-prone

• High-level models: PGI, HMPP, OpenACC
  • High level directives: simplify GPU programming
  • Hiding low-level details from the programmer - main goals of abstraction
  • Reduce learning curve and development time
Related Work

- **hiCUDA, Mint**: automatically translate C code to CUDA code
- **CUDA-lite**: apply global memory optimization via annotations, but it needs separate CUDA kernel functions
- **OpenMPC, OMPCUDA**: source-to-source translation of OpenMP program to CUDA program
Overview of Directive-based Models

HMPP: Hybrid Multicore Parallel Programming workbench

- Two main directives: codelet and callsite
  - Codelet: the function that will be offloaded to accelerator
  - Callsite: the place to call the codelet

- Region directive is the combination of codelet and callsite

- Different codelets can be grouped together to share data

- A set of directives to enhance code generation

- Support multi-GPUs programming
Overview of directive-based models

PGI Accelerator Programming Model

- A set of directives
  - **Compute directive** specifies a portion of the program to be offloaded to accelerator
  - **Loop mapping directive** maps loop parallelism in a fine-grained manner. Two level parallelism: parallel and vector
  - **Data directive** is used to optimize data transfer
- Runtime library routines
- Environment variables
Overview of directive-based models

OpenACC

- Establishes a standard for directive-based accelerator programming
- Contains directives, runtime library routines and environment variables
- Similar to PGI accelerator programming model
- Two types of compute directives: "parallel" and "kernels"
- Three levels parallelism: gang, worker and vector
Experimental Setup

- Evaluate HMPP, PGI and OpenACC for three scientific applications
- GCC 4.4.7 for all sequential programs as well as for HMPP host compiler, -O3 optimization flag

**Table:** Specification of experiment machine

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>Intel Xeon x86_64</td>
</tr>
<tr>
<td>Cores</td>
<td>16</td>
</tr>
<tr>
<td>CPU frequency</td>
<td>2.27GHz</td>
</tr>
<tr>
<td>Main memory</td>
<td>32GB</td>
</tr>
<tr>
<td>GPU Model</td>
<td>Tesla C2075</td>
</tr>
<tr>
<td>GPU cores</td>
<td>448</td>
</tr>
<tr>
<td>GPU clock rate</td>
<td>1.15GHz</td>
</tr>
<tr>
<td>GPU global &amp; constant memory</td>
<td>5375MB &amp; 64K</td>
</tr>
<tr>
<td>Shared memory per block</td>
<td>48KB</td>
</tr>
</tbody>
</table>
2D Heat Conduction

- Formula:

\[
\frac{\partial T}{\partial t} = \alpha \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right)
\]
2D Heat Conduction

- The kernel that does temperature updating is executed on GPU

**Optimizations**

- Loop collapse in kernel region
- Data transfer optimization: make pointer swapping operation occur only in GPU
- Mirror directive in HMPP
- Deviceptr in PGI and OpenACC
- Disable FMA to maintain same computation strategy in different implementations
2D Heat Conduction

**Figure:** 2D Heat Conduction Speedup
FDK Algorithm

FDK: Feldkamp-Davis-Kress

- CT is widely used in medical industry to produce tomographic image of specific area of human body
- FDK is one of the popular 3D-object reconstruction techniques used in CT
- Complexity: $O(N^4)$, where $N$ is the number of detector pixels in one dimension
- Implementation:
  - Code is restructured so that the outmost three loops are tightly nested and collapsed
  - The innermost loop is sequentially executed by every thread
  - Data transfer optimization to remove unnecessary data transfer
- Data: 3D Shepp-Logan head phantom data
- Input: 300 detected images and the resolution of each image is 200*200
- Output: 200*200*200 reconstructed cube
FDK Algorithm

Figure: FDK Speedup with Different Models
CLEVER Algorithm

CLEVER: CLustEring using representatiVEs and Randomized hill climing

- A prototype-based clustering algorithm that seeks for clusters maximizing a plug-in fitness function
- It constructs clusters by seeking an optimal set of representatives one for each clusters; clusters are then created by assigning objects in the dataset to the closest cluster representatives.
CLEVER Algorithm

Implementation:

- Code is converted from C++ to C for better compilation
- Profiling result shows the most time consuming part is the part that assigns objects to the closest representative which computes and compares a lot of distances
- The user-defined structure of dataset and the pointer operation are too complicated to be parsed by compiler. So the code needs to be restructured.
- The whole dataset is read only, so it will stay in GPU global memory during execution
### CLEVER Algorithm

#### Table: L10Ovals Dataset Characteristics

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>335,900 objects</td>
</tr>
<tr>
<td>Attributes</td>
<td>(&lt;x, y, \text{class label}&gt;)</td>
</tr>
<tr>
<td>Distance Function</td>
<td>Euclidean Distance</td>
</tr>
<tr>
<td>Plug-in Fitness Function</td>
<td>Purity: Percentage of objects belonging to the majority class of the cluster</td>
</tr>
</tbody>
</table>

#### Table: Earthquake Dataset Characteristics

<table>
<thead>
<tr>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data size</td>
<td>330,561 objects</td>
</tr>
<tr>
<td>Attributes</td>
<td>(&lt;\text{latitude, longitude, depth}&gt;)</td>
</tr>
<tr>
<td>Distance Function</td>
<td>Euclidean Distance</td>
</tr>
<tr>
<td>Plug-in Fitness Function</td>
<td>High Variance: Measures how far the objects in the cluster are spread out with respect to earthquake depth</td>
</tr>
</tbody>
</table>
Figure: CLEVER Speedup with Different Models
Summary

Table: Time (in sec) consumed by serial, CUDA, HMPP, PGI and OpenACC versions of the code, only for most time-consuming dataset

<table>
<thead>
<tr>
<th>Applications</th>
<th>Serial</th>
<th>CUDA</th>
<th>HMPP</th>
<th>PGI</th>
<th>OpenACC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>2D Heat</td>
<td>8922.81</td>
<td>59.13</td>
<td>60.78</td>
<td>72.74</td>
<td>75.65</td>
</tr>
<tr>
<td>CLEVER</td>
<td>116.15</td>
<td>23.04</td>
<td>25.08</td>
<td>101.51</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>73.31</td>
</tr>
</tbody>
</table>
Conclusion and Future Work

• Conclusions:
  • High-level models provide a high-level abstraction by hiding most of the low-level complexities of the GPU platform
  • The performance is highly dependent on the application characteristics.
  • Directive-based models can achieve around 80% and sometimes more than 90% performance of CUDA code
  • OpenACC is still being constructed and may require fine tuning

• Future Work:
  • Multi-GPUs support in OpenACC
  • Add more loop optimization clauses in OpenACC