Heterogeneous Computing With GPGPUs

Matthew Piehl
Overview

• Introduction to CUDA
• Project Overview
  – Issues faced
  – nvcc
  – Implementation
• Performance Metrics
• Conclusions
  – Improvements
  – Advantages
  – Disadvantages
Introduction to CUDA

- Copy data from Main Memory to GPU Memory
- CPU instructs process to GPU
- GPU Executes on many cores
- Transfer Data back to Main Memory
Introduction to CUDA

- Nvidia Tesla S1070
  - 4 Graphics cards
  - 4 x 240 thread processors (TPs)
    - 960 total TPs
  - 4Gb Memory
The Project

• Offload the Boolean conjugation operations such as AND and OR operators to GPU
  - In theory performance should improve since graphics cards specialize in Boolean operations (ANDing and ORing)
CUDA Based Approach

- With PGI not an option, I have to move to Nvidia's compiler nvcc
- Nvcc is Nvidia's CUDA preprocessor.
  - nvcc only compiles your kernel code
  - Passes your device (C code) to gcc
- A namespace nightmare with PGI
  - nvcc -> gcc -> pgCC
CUDA Based Approach

- Since nvcc only compiles C code we need an external C function
- cudaMalloc()
- cudaMemcpy()  
  - Host to Device
  - Device to Host
CUDA Based Approach

/*Allocate memory on Nvidia Device*/

cudaMalloc((void**) &Ptrinc, memsize);
cudaMalloc((void**) &Ptrcurr, memsize);
cudaMalloc((void**) &Ptrtmp, memsize);

/*Copy to allocated Memory*/

cudaMemcpy(Ptrinc, incoming, memsize, cudaMemcpyHostToDevice);
cudaMemcpy(Ptrcurr, curr, memsize, cudaMemcpyHostToDevice);
offload_OR<<<1,wordcnt>>>(Ptrinc,Ptrcurr,Ptrtmp);
CUDA Based Approach

```c
__global__ void offload_OR(size_t* inc, size_t* curr, size_t* tmp)
{
    int i = threadIdx.x;
    tmp[i] = inc[i] | curr[i];
}
```
CUDA Based Approach

cudaMemcpy(newtree,Ptrtmp,memsize,cudaMemcpyDeviceToHost);

/*Free our Nvidia allocated memory*/
cudaFree(Ptrinc);
cudaFree(Ptrtmp);
cudaFree(Ptrcurr);

return newtree;
<table>
<thead>
<tr>
<th>Wordcnt</th>
<th>GPU Time (ms)</th>
<th>Total GPU Time with Memory Latency (ms)</th>
<th>CPU Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (8Bytes)</td>
<td>24</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>2 (16Bytes)</td>
<td>23</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>16 (128Bytes)</td>
<td>27</td>
<td>74</td>
<td>2</td>
</tr>
<tr>
<td>157 (1.22KB)</td>
<td>30</td>
<td>78</td>
<td>3</td>
</tr>
<tr>
<td>1563 (12KB)</td>
<td>56</td>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>15625 (122KB)</td>
<td>57</td>
<td>257</td>
<td>8</td>
</tr>
<tr>
<td>156250 (1.19MB)</td>
<td>65</td>
<td>1565</td>
<td>368</td>
</tr>
<tr>
<td>1562500 (11.9MB)</td>
<td>76</td>
<td>9836</td>
<td>3485</td>
</tr>
<tr>
<td>15625000 (119MB)</td>
<td>78</td>
<td>94037</td>
<td>45181</td>
</tr>
<tr>
<td>78125000 (596MB)</td>
<td>80</td>
<td>390958</td>
<td>251301</td>
</tr>
<tr>
<td>1562500000 (1.19GB)</td>
<td>81</td>
<td>949187</td>
<td>389447</td>
</tr>
</tbody>
</table>
Conclusion

- Memory latency causing problems
  - Spending way too much time moving data to and from the device
- Current program is using limited loop vectorization
  - CUDA implementation is dispatching 1000s of threads
    - 960 thread processors to work on
    - Very little overhead in thread creation
Conclusions

- Can we get around this issue of memory latency?
  - Current environment isn't enough
  - A heterogeneous cluster would have potential
    - Build the PTrees on the device don't copy back
      - 4GB limit on each device
    - 2-Dimensional PTrees can be helpful
Conclusions

- General disadvantages of CUDA
  - Memory latency a major issue
  - 4Gb of device memory
  - Steep learning curve vs PGI
  - Linking and namespace nightmare when linking many different compilers together
Conclusions

- Several advantages of CUDA
  - Very good when you have to do many computations with limited memory copies
    - Integral approximation (Monte Carlo Simulation, Simpsons Rule)
  - Supports thousands of threads