Enhancing productivity on heterogeneous supercomputers with task-based programming model
Adrien Roussel, Mickael Boichot, Romain Pereira, Manuel Ferat

To cite this version:
Adrien Roussel, Mickael Boichot, Romain Pereira, Manuel Ferat. Enhancing productivity on heterogeneous supercomputers with task-based programming model. SIAM CSE 2023 - SIAM Conference on Computational Science and Engineering, Feb 2023, Amsterdam, Netherlands. cea-03994014

HAL Id: cea-03994014
https://cea.hal.science/cea-03994014
Submitted on 28 Feb 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Enhancing Productivity on Heterogeneous Supercomputers with Task-based Programming Model

Adrien Roussel, Mickael Boichot, Manuel Ferat, Romain Pereira
CEA, DAM, DIF

SIAM CSE, Amsterdam
February 28, 2023
Most of the time, scientific applications are already parallelized for CPU (or partly for GPU).

To take advantage of GPUs, developers have to rewrite their codes (in part? in whole?).

**Goals**:  
- Build a preliminary analysis to determine the potential of the application for a GPU port  
- Port mini-application to heterogeneous architectures with this information
Contributions

Goal: Inform the user

- that there exists a **test case** of the application **suitable** for a **GPU port**
- about the **strengths** and **weaknesses** of the application for a **GPU port**

- We analyzed the variations of two **metrics**:
  - the **degree of parallelism** (→ saturate GPU cores)
  - and the **working set** (→ bounded GPU memory)

```c
int main(...) {
    /* parallel CPU code */
}
```

- Porting app. To GPU Computing
  - Using Task based prog. Model
    - MPI+OpenMP (tasks + target)

---

*(Note: Image not transcribed to text for the diagram.)*
Contents

1. Introduction

2. Application Characterization Method

3. Proxy Applications Analysis

4. Porting Proxy App to GPU: LULESH

5. Conclusion
2. Application Characterization Method
Targeted GPU Characterization

Maximizing FFMA throughput ("floating point fused multiple add")

\[ \text{RP} = \#\text{cores} \cdot L \]

where

- \( L \): 32 bits FFMA instruction latency

Required Parallelism (RP)

Characterization

Targeted GPU

L2 Cache

DRAM

RAM (GB)

Already known (GPU documentation)
Applications analyzed at Loop Granularity (already parallelized with OpenMP)

L: for (i = 0; i < N; ++i) compute;

- Set of data necessary to run the loop
- Working Set
- Parallelism Degree $P(L)$

- $# \text{ independent Iterations (i.e., tripcount)}$
- $\text{Working Set } WS_{\text{malloc}}(L)$
- $\text{Working Set } WS_{\text{cache line}}(L)$
- $\text{measured at two Granularities, i.e., the size of smallest accessed memory element (using DynamoRIO)}$
Extrapolation Methodology

For each loop

\begin{align*}
\forall L \in S, \ n \in \{n_1, n_2, \ldots\} \\
\rightarrow \text{measure} \\
\begin{itemize}
  \item working set \ WS(L)
  \item parallelism degree \ P(L)
\end{itemize}
\end{align*}

\begin{align*}
\forall L \in S, \\
WS(L) = f(P(L)) \\
P(L) = g(n)
\end{align*}

\begin{tabular}{|c|c|c|c|}
\hline
loop & working set & parallelism degree & input size \\
\hline
L & WS(L) & P(L) & n_1 \\
L & WS(L) & P(L) & n_2 \\
... & ... & ... & ... \\
\hline
\end{tabular}
Extrapolation Methodology

Extract most time-consuming loops

∀ L ∈ S, n ∈ \{n1, n2, ...\} → measure
- working set WS(L)
- parallelism degree P(L)

∀ L ∈ S, WS(L) = f(P(L))
\[ P(L) = g(n) \]

projection

Test cases fitting target GPU memory and exposing enough parallelism
3. Proxy Applications Analysis

LULESH, miniFE
Experimental Protocol

- **Evaluation Machine**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>processor</td>
<td>Intel Xeon E5-2680</td>
<td>#sockets</td>
<td>2</td>
</tr>
<tr>
<td>#sockets</td>
<td></td>
<td>#cores/socket</td>
<td>8</td>
</tr>
<tr>
<td>RAM (GB)</td>
<td>128</td>
<td>OS</td>
<td>Linux 3.10</td>
</tr>
<tr>
<td>Compiler</td>
<td>Intel 20.0.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Evaluated mini-applications**

  - **LULESH** (hydrodynamic simulation)
  - **MiniFE** (finite element problem)
  - Quicksilver (Monte Carlo particle transport)
  - XSBench (Monte Carlo particle transport kernel)

*All these applications have CPU OpenMP, GPU CUDA and HIP versions.*

- **Considered targeted GPUs**

<table>
<thead>
<tr>
<th>GPU</th>
<th>RAM</th>
<th>#cores</th>
<th>minimal required parallelism RP</th>
</tr>
</thead>
<tbody>
<tr>
<td>V100</td>
<td>16 GB</td>
<td>5120</td>
<td>20480</td>
</tr>
<tr>
<td>A100 40GB</td>
<td>40 GB</td>
<td>6912</td>
<td>27648</td>
</tr>
<tr>
<td>A100 80GB</td>
<td>80 GB</td>
<td>6912</td>
<td>27648</td>
</tr>
<tr>
<td>MI100</td>
<td>32 GB</td>
<td>7680</td>
<td>30720</td>
</tr>
</tbody>
</table>
LULESH

Working set of LULESH selected loops as a function of their tripcount

Projection of dataset input size $s_{max}$ saturating GPU memory and corresponding $\rho$ ratio ($\rho(L) = P(L) / RP$)

- Sufficient exposed parallelism ($\rho \gg 1$)
- Limiting factor: memory size
- Advised data transfers granularity: malloc

- Similar conclusions for miniFE
4. Porting Proxy App to GPU: LULESH
Porting LULESH to MPI+OpenMP(tasks)

- **LULESH**
  - Hydrodynamic problems solver - *Iterative* simulation
  - Original code express *loop-based parallelism* (parallel-for)

- **Porting to MPI+OpenMP (tasks)**
  - Loops splitted into tasks with dependencies
  - Single producer/multi-consumer scheme
  - MPI communications nested within OpenMP tasks
    - Overlap communications by computations

```
for (int j = 0 ; j < n_nodes; j += BS)
{
    # pragma omp task depend(out : mesh->nodes[j])
    {
        for (int i = j ; i < j + BS ; ++i)
            work_on_nodes(mesh->nodes[i]);
    }
}
```

```
for (int j = 0 ; j < n_cells; j += BS)
{
    # pragma omp task depend(in : [...])
    {
        for (int i = j ; i < j + BS ; ++i)
            work_on_cells(mesh->cells[i]);
    }
}
```
Concept Generalization for Heterogeneous Architectures with OpenMP Target

- Porting to MPI+OpenMP (tasks + GPU target)
  - GPU « super-tasks » merging several CPU tasks
  - From CPU tasks to GPU offloading while enhancing asynchronism
    - Sending/Retrieving data to/from GPU with ‘update’ clause
    - Executing GPU kernels through ‘nowait’ clause
  - Offloading 6 loops given by application characterization
    - IntegrateStressForElems, CalcKinematicsForElems, CalcAccelerationForNodes, CalcFBHourglassForceForElems1, CalcHourglassControlForElems, CalcMonotonicQGradientsForElems
- Use of cudaMallocHost for the GPU buffers
  - Pinned memory to enable asynchronous execution

```c
1 buffer = cudaMallocHost(...);

[...]
2 for (int j = 0 ; j < n_nodes; j += BS)
3 {
4   # pragma omp task depend(out : mesh->nodes[j])
5   {
6     for (int i = j ; i < j + BS ; ++i)
7       work_on_nodes(mesh->nodes[i], buffer[i]);
8   }
9 # pragma omp target 
10   teams distribute parallel for nowait 
11   depend(iterator(i=0:N_BS), out: mesh->nodes[j+i*BS])
12   {
13     for (int i = j ; i < j + N_BS * BS ; ++i)
14       work_on_nodes(mesh->nodes[i], buffer[i]);
15   }
16 # pragma omp target update from(buffer[j:BS]) 
17   depend(out: mesh->nodes[j])
```
## Experimental Protocol

<table>
<thead>
<tr>
<th>Processor</th>
<th>AMD EPYC 7H12</th>
</tr>
</thead>
<tbody>
<tr>
<td>#sockets</td>
<td>2</td>
</tr>
<tr>
<td>#cores/socket</td>
<td>64</td>
</tr>
<tr>
<td>GPU</td>
<td>NVIDIA A100</td>
</tr>
<tr>
<td>#GPU/socket</td>
<td>2</td>
</tr>
<tr>
<td>RAM (GB)</td>
<td>256</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPC-OpenMP</th>
<th>4.1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLVM (Offloaded part)</td>
<td>16.x</td>
</tr>
<tr>
<td>Open MPI</td>
<td>4.0.5</td>
</tr>
</tbody>
</table>
LULESH Performance Evaluation

- **Problem size**
  - 264³ elements
  - 80% of the NUMA domain memory capacity
  - 20% of the GPU memory capacity
  - 128 iterations

- **Placement**
  - Per MPI Process:
    - 16 threads (on 16 cores)
    - + 1 GPU

- **Weak scaling**
  - Median runtime of 10 runs
  - Each version scale very well
  - **OpenMP tasks** version significantly improves performances due to **cache reuse**
  - Significant performance gain with **GPU offloading**

<table>
<thead>
<tr>
<th>ranks</th>
<th>parallel-for</th>
<th>no-offloading</th>
<th>offloading</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>565.84</td>
<td>325.02</td>
<td>209.31</td>
</tr>
<tr>
<td>8</td>
<td>572.45</td>
<td>331.09</td>
<td>209.43</td>
</tr>
<tr>
<td>27</td>
<td>581.26</td>
<td>333.98</td>
<td>212.78</td>
</tr>
</tbody>
</table>
5. Conclusion
Conclusion

- An approach: (int main(...){
    ...
}) \mapsto \{ \text{test cases such that int main(...){
    ...
}} \text{fit on RAM}

and do not lack parallelism

- complementary to existing work (analysis of several application runs \(\rightarrow\) extrapolation)

- We have evaluated our approach and shown that we can qualitatively guide developer in their GPU porting

- We have ported LULESH on heterogeneous machine with a task-based programming model
  - Significant gains are observed
Thank you!

Questions?