CPU+GPU Load Balance Guided by Execution Time Prediction

Jean-François Dollinger, Vincent Loechner

Inria CAMUS, ICube Lab., University of Strasbourg

jean-francois.dollinger@inria.fr, vincent.loechner@inria.fr

19 January 2015
1. Introduction

2. Prediction
   - Overview
   - Code generation
   - Profiling

3. Runtime
   - CPU + GPU

4. Conclusion
Achieving and predicting performance on CPU/GPU is difficult. Sensitive to:

- Input dataset (CUDA grid size, cache effects)
- Compiler optimizations (unrolling, fission)
- Cloudy infrastructures
- Hardware availability
- Efficient resources exploitation
Because of dynamic behaviors compilers miss performance opportunities

- PLUTO
- PPCG
- Par4All
- openACC/HMPP: manual tuning

→ Automatic methods are the way to go (typical use case)
→ Our interest: polyhedral codes
How to get performance?

- Right code with right PU (Processing Unit)
- Select best code version on each given PU
- Ensure load balance between PUs

→ Multi-versioning + runtime code selection = win
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU + GPU

4 Conclusion
Multi-versioning: performance factors

- Static factors (instruction)
- External dynamic factors (scheduler)
- Internal dynamic factors (cache effects, memory contention)
Static code generation

```
#pragma scop
for(;
    for(;
        for(;
            S0(...);
#pragma endscop
```

Offline profiling

```
Launch PPCG
Launch PLUTO
Build templates
```

Runtime prediction

```
Memcpy duration
Kernel duration
GPU
CPU
Schedule
```

Application binary object file
```
... /* scop */
call schedule(...)
call dispatch(...)
... /* endscop */
... ```
Pedro Framework [Benoit Pradelle et al. 2011]

- Multi-versioning of polyhedral loop nests
- Target: multicore CPUs
Code version

- Block size
- Tile size
- Schedule

→ controlled by PPCG cmd line options

PPCG, source-to-source compiler

- Transforms C to CUDA
- Generates:
  - Ehrhart polynomials
  - Loop nest parameters

Python scripts

- Fill templates in C code
1. Introduction

2. Prediction
   - Overview
   - Code generation
   - Profiling

3. Runtime
   - CPU + GPU

4. Conclusion
Data transfers: host ↔ device
- Parameter: message size
- Asymmetric and non-uniform bandwidth

Code simulation
- Parameters: number of CUDA blocks, sequential parameters
- Load balance
- Memory contention

How to model the performance curves?
- Affine intervals detection
1st test platform
   • 2 Nvidia GTX 590 (16 (SM) * 32 (SP))
   • Asus P8P67-Pro (PCle 2, x8 per card)
   • Core i7 2700k, stock

2nd test platform
   • Nvidia GTX 680 (8 (SM) * 192 (SP))
   • Asus P8P67-Deluxe (PCle 2, x16)
   • Core i7 2600
Prediction

Data transfers (testbed 1)
Prediction

Data transfers (testbed 2)
Kernel simulation (testbed 1)

Prediction

gemm 32x16 - GTX 590

execution time per iteration (ns)
number of blocks

real
profiled

0.01
0.1
1
10
100
101
102
103
104
105

10^0
10^1
10^2
10^3
10^4
10^5

Prediction
Kernel simulation (testbed 1)

syrk2 - GTX 590

\[ e_i = p_i \beta + u_i \]
Outermost parallel loop split into chunks
- Each chunk associated to one PU
- PUs performance differ

→ Ensure load balance

Multi-Versioning
- Code optimized towards target (PLUTO + PPCG)
- Multiple code versions (combined)

Two components:
- Scheduler:
  - Execution time of chunks [B. Pradelle et al.] + [J-F. Dollinger et al.]
  - Adjust chunks sizes
- Dispatcher
Scheduler functioning

1. \[ T_0 = t_0 \times \text{Card } D_0 \approx t_1 \times \text{Card } D_1 \approx \ldots \approx t_n \times \text{Card } D_n \]
2. \[ T_i \text{ must tend to } \frac{1}{n} \sum_{i=0}^{n-1} (t_i \times \text{Card } D_i) = \frac{1}{n} \times T_{all} \]
3. \[ t_i = f(G_i, \text{seq}) \text{ on GPU} \]
4. \[ t_i = g(P_i, S_i) \text{ on CPU} \]

Eliminate inefficient PUs
Runtime
CPU + GPU (speedup to one PU)

diagram showing speedup comparison for different configurations: CPU, 1GPU, CPU+1GPU, CPU+2GPUs, CPU+3GPUs, and CPU+4GPUs. The x-axis represents different operations (gemm, 2mm, 3mm, syrk, syr2k, doitgen, gesummvmvt, gemver), and the y-axis represents speedup. The legend indicates different color codes for each configuration.
Runtime
Multiversioning CPU + GPU (speedup to worst)
Framework capabilities

- Execution time prediction
- Fastest version selection
- CPU vs GPU competition
- CPU + GPU joint usage

Future work

- Energy consumption
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU + GPU

4 Conclusion
Offline profiling: ranking table

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>version 1</th>
<th>version 2</th>
<th>version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40 ms</td>
<td>55 ms</td>
<td>32 ms</td>
</tr>
<tr>
<td>2</td>
<td>32 ms</td>
<td>28 ms</td>
<td>17 ms</td>
</tr>
<tr>
<td>3</td>
<td>22 ms</td>
<td>15 ms</td>
<td>9 ms</td>
</tr>
<tr>
<td>4</td>
<td>14 ms</td>
<td>7 ms</td>
<td>8 ms</td>
</tr>
</tbody>
</table>

Online prediction: execution time computation

\[
observation = \{2000, 600, 300, 300\}
\]
\[
prediction (version1) = ((2000 - 600) \times 40) + ((600 - 300) \times 32) + (0 \times 22) + (300 \times 14)
\]
\[
= 69800ms
\]
The algorithm stages:

- Init.: distribute iterations equitably amongst PUs
- Repeat 10 times:
  - Compute per chunk execution time
  - $r_i = T_i / T_{all}$
  - Adjust chunk size according to $r_i$