CPU + GPU load balancing guided by execution time predictions

Jean-François Dollinger, Vincent Loechner

Inria CAMUS, ICube Lab., University of Strasbourg

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

4th December 2013
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU vs GPU
   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, nest 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
→ Our interest: polyhedral codes
How to get more performance?

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

→ Multi-versioning + runtime code selection = win
Outline

1. Introduction

2. Prediction
   - Overview
   - Code generation
   - Profiling

3. Runtime
   - CPU vs GPU
   - CPU + GPU

4. Conclusion
Multi-versioning: performance factors

- Static factors (instruction)
- External dynamic factors (scheduler)
- Internal dynamic factors (cache effects, memory contention)
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU vs GPU
   CPU + GPU

4 Conclusion
Code version

- Block size
- Tile size
- Schedule

PPCG, source-to-source compiler

- Transforms C to CUDA
- Generates:
  - Ehrhart polynomials
  - Sequential and Parallel parameters

Python scripts

- Fill templates in C code
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU vs GPU
   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

Optimization
  • Affine intervals detection
1st test platform
  • 2 Nvidia GTX 590 (16 (SM) * 32 (SP))
  • Asus P8P67-Pro (PCIe 2, x8 per card)
  • Core i7 2700k, stock

2nd test platform
  • Nvidia GTX 680 (8 (SM) * 192 (SP))
  • Asus P8P67-Deluxe (PCIe 2, x16)
  • Core i7 2600
Data transfers (testbed 1)

![Graph showing data transfers with different test configurations]

- real dev-host GTX590
- real host-dev GTX590
- prof. dev-host GTX590
- prof. host-dev GTX590
Prediction
Data transfers (testbed 2)

![Graph showing data transfers](image-url)
Prediction
Kernel simulation (testbed 1)

gemm 32x16 - GTX 590

execution time per iteration (ns)
number of blocks
gemm 32x16 - GTX 590
real
profiled

10 / 32
Prediction
Kernel simulation (testbed 1)

\[ e_i = p_i \beta + u_i \]
"Fastest wins"

- Run codes concurrently (CPU and GPU)
- Winner stops the other codes

CPU code interruption ingredients

- Extract OpenMP *parallel for* regions
- Mix pthread and OpenMP
  - Thread ID with `pthread_self`
  - Signal with `pthread_kill`
- Save/restore context with `setjmp/longjmp`
- Check flag after *parallel for* region
GPU code interruption ingredients

- **Host code**
  - Kernel call and transfers enqueued in `stream1`
  - Check interruption flag after `cudaMemcpyAsync`
  - Check interruption flag after kernel synchronization call

- **Device code**
  - *Global* GPU variable specifies behavior
  - Poll variable inside second loop level
  - If(variable == 1) call trap instruction

CPU issues `cudaMemcpyAsync(..., stream2)` to modify polled variable
Runtime

CPU vs GPU (stop overhead standard dataset)

![Bar chart showing runtime comparison between CPU and GPU for various benchmarks. The chart displays overhead values for different kernels and linear algebra operations, with notable peaks for specific benchmarks.]
Runtime

CPU vs GPU (stop overhead large dataset)
Puzzle game:

- *trmm*, third loop is parallel
- CPU alone: 15 seconds
- CPU vs GPU, CPU side: 7 seconds
- Why?
1 Introduction

2 Prediction
   Overview
   Code generation
   Profiling

3 Runtime
   CPU vs GPU
   CPU + GPU

4 Conclusion
Outermost parallel loop split into chunks

- Each chunk associated to one PU
- PUs performance differ

→ Ensure load balance

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. $T_i$ must tend to $1/n \times \sum_{i=0}^{n-1} (t_i \times \text{Card } D_i) = 1/n \times T_{all}$

3. $t_i = f(G_i, \text{seq})$ on GPU

4. $t_i = g(P_i, S_i)$ on CPU

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 $r_i$
Runtime

CPU + GPU (execution time)
Runtime
CPU + GPU (load imbalance)
Conclusion

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

Future work
- Finish the automatic code generation
- Defend PhD (2014)
Outline

1. Introduction
2. Prediction
   - Overview
   - Code generation
   - Profiling
3. Runtime
   - CPU vs GPU
   - CPU + GPU
4. Conclusion

Thanks! Answer to puzzle game