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

Outline

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?

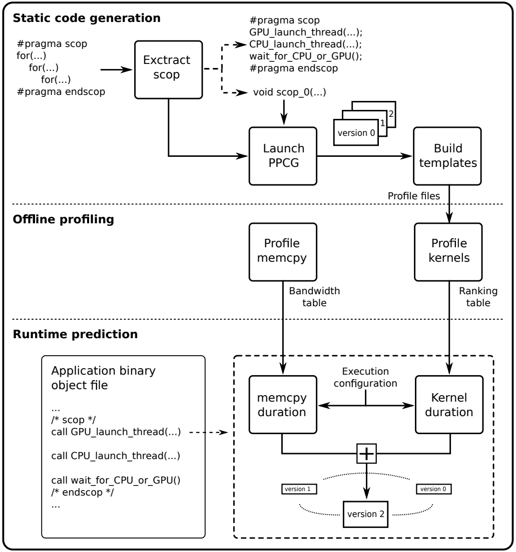
- Right code with right PU (Processing Unit)
- Select PU best code version
- Ensure load balance between PUs

$\rightarrow$ Multi-versioning + runtime code selection = win

Outline

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(...)
#pragma endscop
Exctract scop
#pragma scop
GPU_launch_thread(...);
CPU_launch_thread(...);
wait_for_CPU_or_GPU();
#pragma endscop
void scop_0(...)
Launch PPCG
version 0
1
2
Build templates
Profile files
Offline profiling
Profile memcpy
Profile kernels
Bandwidth table
Ranking table
Runtime prediction
Application binary object file
...
/* scop */
call GPU_launch_thread(...)
call CPU_launch_thread(...)
call wait_for_CPU_or_GPU()
/* endscop */
...
Execution configuration
memcpy duration
Kernel duration
+
version 1
version 0
version 2

Outline

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

Data transfers: host $\leftrightarrow$ device

- Parameter: message size
- Asymetric and non-uniform bandwidth

Code simulation

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

Optimization

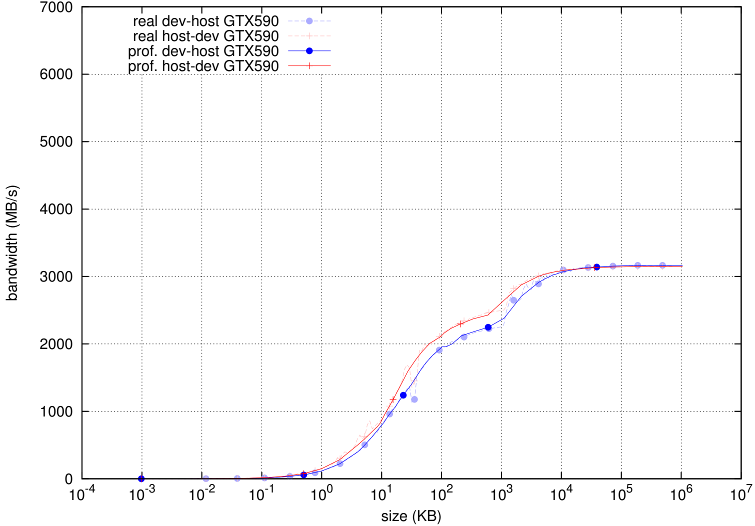
- Affine intervals detection

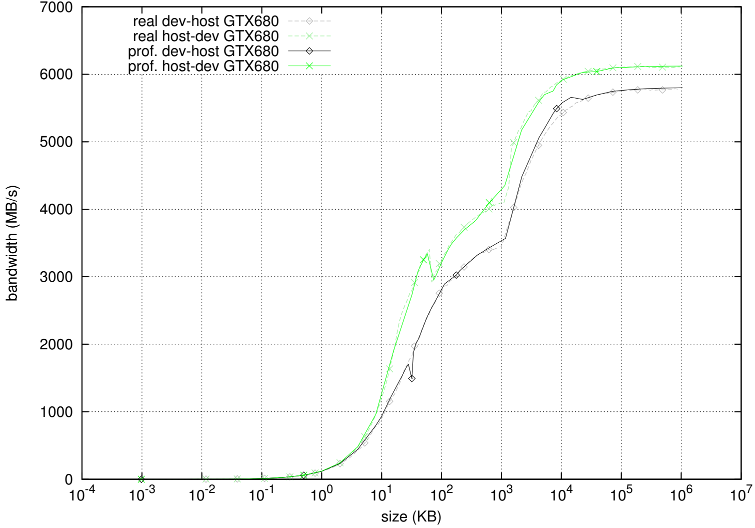
1st test platform

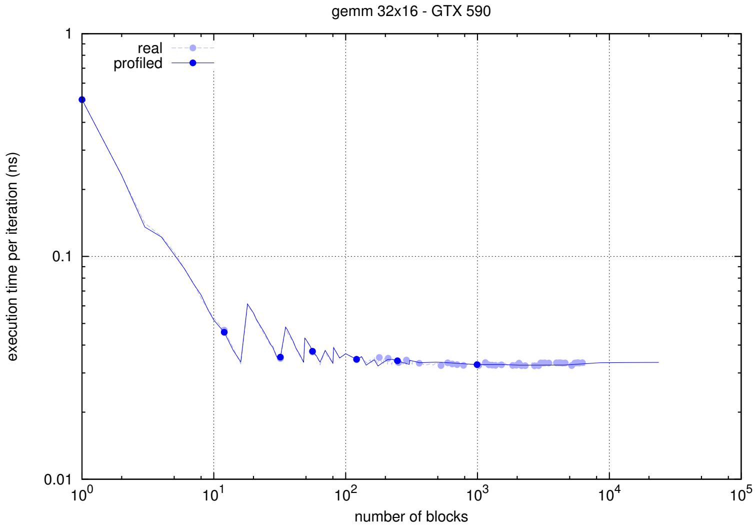
- 2 Nvidia GTX 590 (16 (SM) * 32 (SP))
- Asus P8P67-Pro (PCIe 2, x8 per card)
- Core i7 2700k, stock

2nd test platform

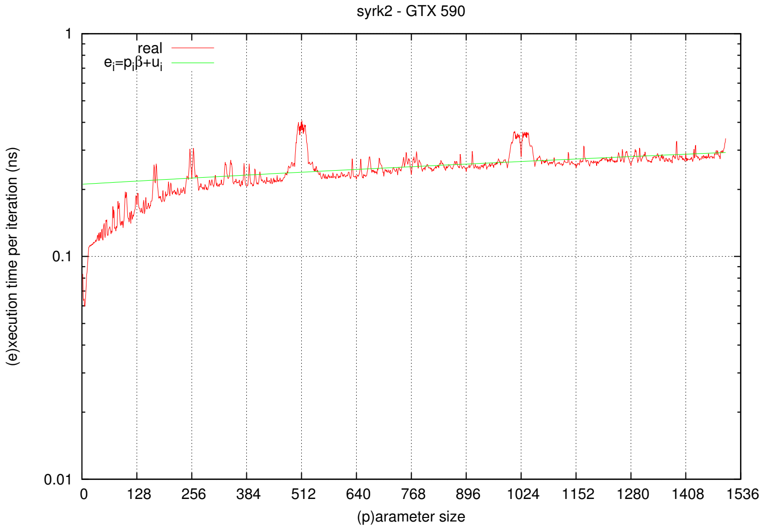
- Nvidia GTX 680 (8 (SM) * 192 (SP))
- Asus P8P67-Deluxe (PCIe 2, x16)
- Core i7 2600


real dev-host GTX590
real host-dev GTX590
prof. dev-host GTX590
prof. host-dev GTX590
bandwidth (MB/s)
7000
6000
5000
4000
3000
2000
1000
0
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
size (KB)


real dev-host GTX680
real host-dev GTX680
prof. dev-host GTX680
prof. host-dev GTX680
bandwidth (MB/s)
size (KB)
0
1000
2000
3000
4000
5000
6000
7000
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}



gemm 32x16 - GTX 590
real
profiled
execution time per iteration (ns)
1
0.1
0.01
10^0
10^1
10^2
10^3
10^4
10^5
number of blocks


syrk2 - GTX 590
real
$e_i=p_i\beta+u_i$
(e)xecution time per iteration (ns)
1
0.1
0.01
0
128
256
384
512
640
768
896
1024
1152
1280
1408
1536
(p)arameter size

Outline

"Fastest wins"

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

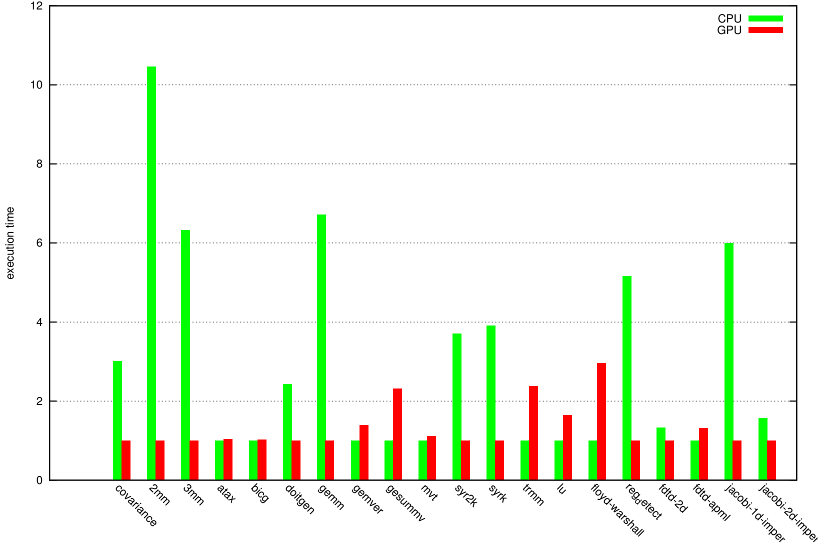
CPU code interruption ingredients

- 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

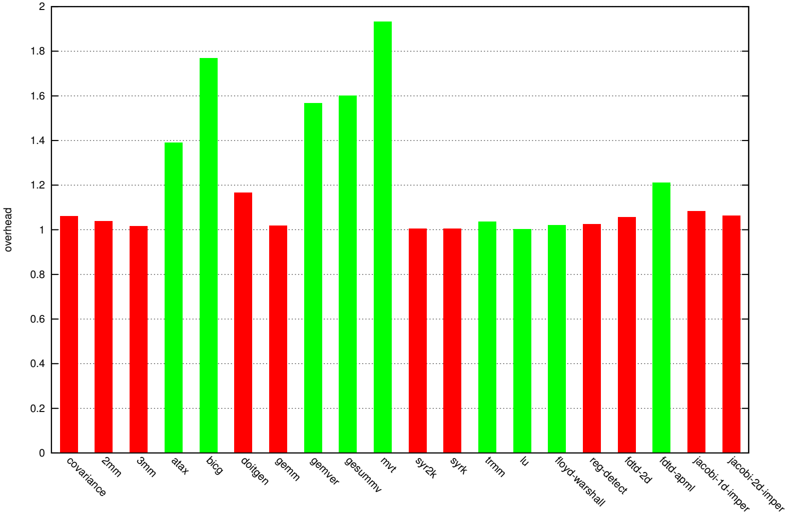
GPU code interruption ingredients

- 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

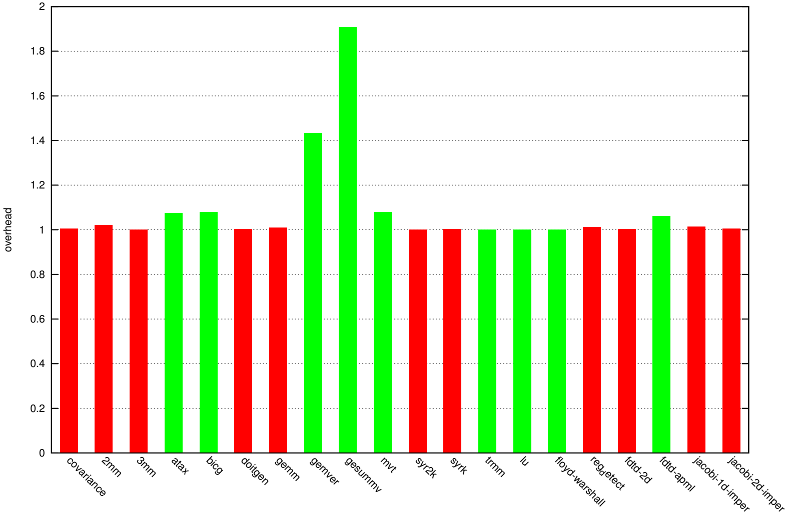

12
10
8
6
4
2
0
execution time
CPU
GPU
covariance
2mm
3mm
atax
bicg
doitgen
gemm
gemver
gesummv
mvt
syr2k
syrk
trmm
lu
floyd-warshall
regdetect
fdtd-2d
fdtd-apml
jacobi-1d-imper
jacobi-2d-imper

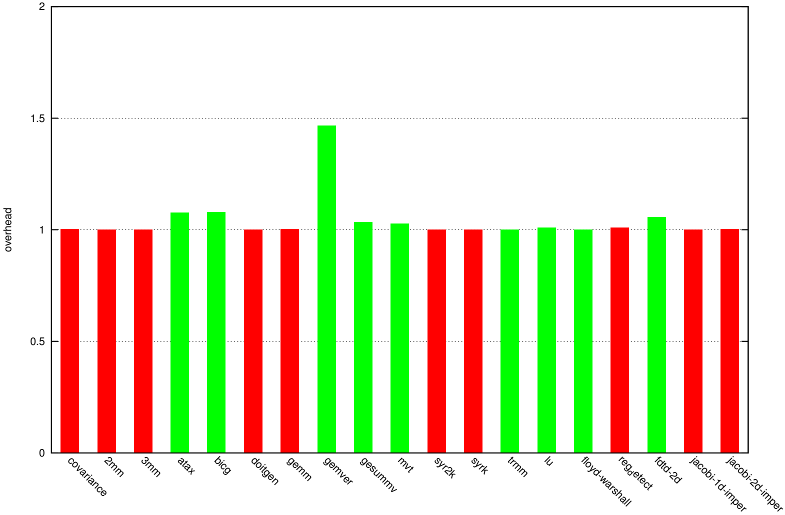
Runtime
CPU vs GPU (overhead)



Runtime

CPU vs GPU (stop overhead standard dataset)




2
1.5
1
0.5
0
overhead
covariance
2mm
3mm
atax
bicg
doitgen
gemm
gemver
gesummv
mvt
syr2k
syrk
trmm
lu
floyd-warshall
reg_detect
fdtd-2d
jacobi-1d-imper
jacobi-2d-imper

Puzzle game:

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

Outline

Outermost parallel loop split into chunks

- Each chunk associated to one PU
- PUs performance differ

→ Ensure load balance

Two components:

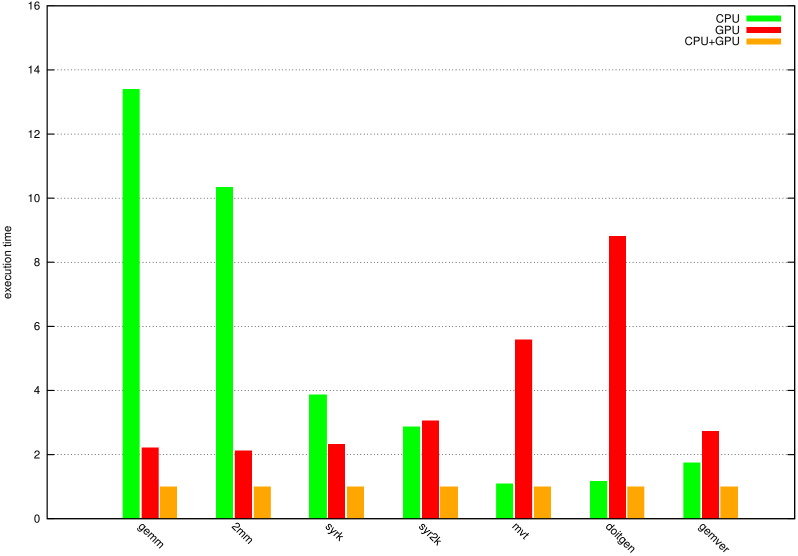
- Scheduler:
    - Execution time of chunks [B. Pradelle et al.] + [J-F. Dollinger et al.]
    - Adjust chunks sizes
- Dispatcher

Scheduler functioning

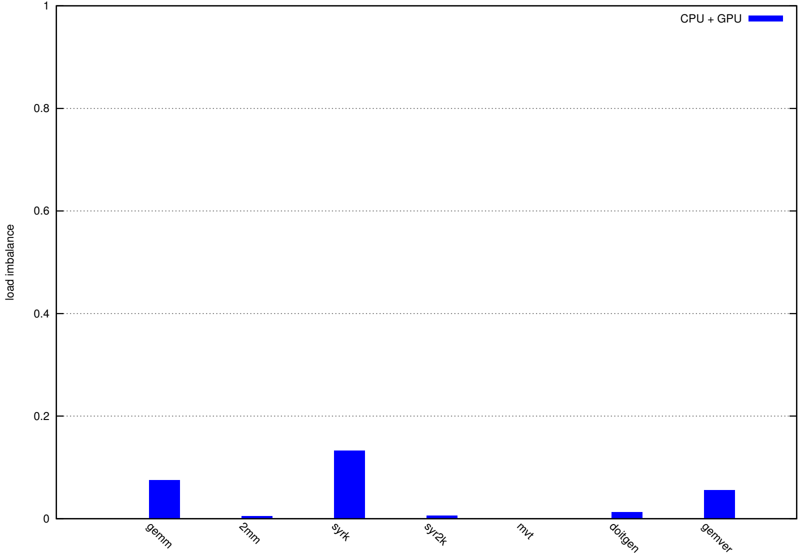
1. $T_0 = t_0 * Card\ D_0 \approx t_1 * Card\ D_1 \approx ... \approx t_n * Card\ D_n$
2. T_i must tend to $1/n * \sum_{i=0}^{n-1}(t_i * Card\ D_i) = 1/n * T_{all}$
3. $t_i = f(G_i, \overline{seq})$ on GPU
4. $t_i = g(P_i, S_i)$ on CPU

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

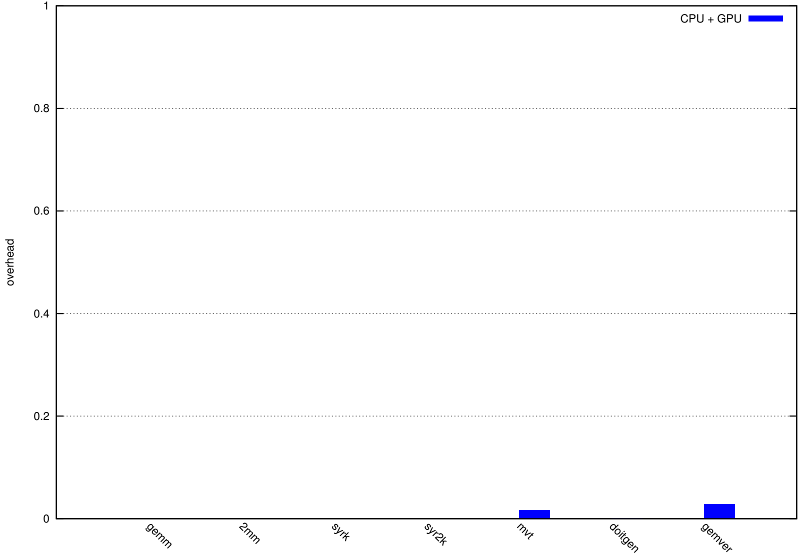

16
14
12
10
8
6
4
2
0
execution time
CPU
GPU
CPU+GPU
gemm
2mm
syrk
syr2k
mvt
doitgen
gemver

Runtime
CPU + GPU (load imbalance)



Runtime

CPU + GPU (overhead)



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

Thanks! **Answer to puzzle game**