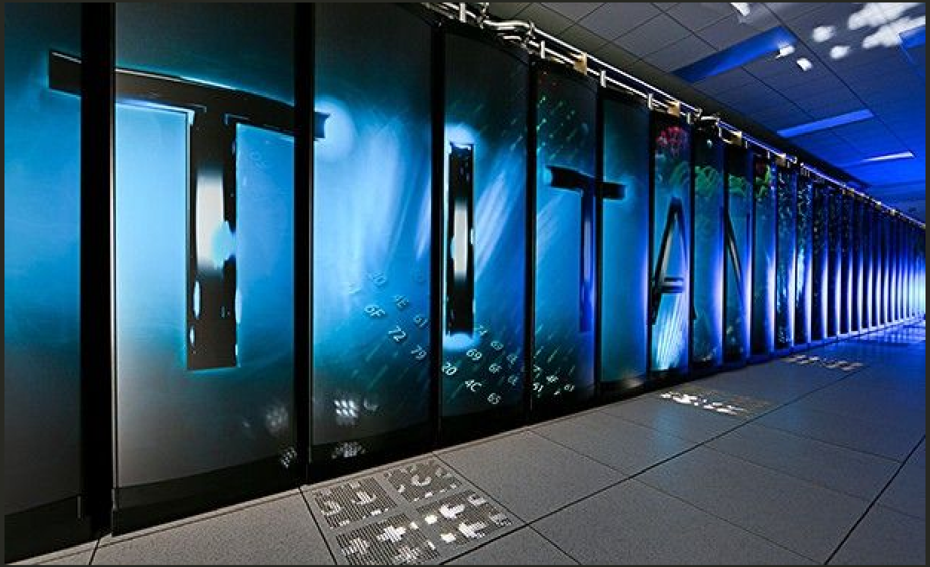
Runtime support for approximate computing on heterogeneous systems.

Spyrou Michalis
MSc Thesis
University of Thessaly

Energy efficiency



Heterogeneous systems

Increased development effort:

- Multiple address spaces.
- Scheduling.
- Hard to utilize every resource.
- Even OpenCL is hard to manage (multiple kernel/memory/program objects, command queues etc.)

Approximate computing

Some portions of an application are less significant than others.

Substitute calculations with "cheaper" ones, fixed values or even drop.

SPStereo Disparity heatmap: Accurate vs Approximate

Our goals

Reduce energy consumption by executing computations at lower accuracy.

Remove common programming burdens (data management, scheduling).

Unified run-time support for heterogeneous systems.

Concurrently exploit all available resources of a heterogeneous system.

Real time power and energy monitoring.

Background

Code example - DCT

```
__kernel void dctAccurate( double *image, double *result, int subblock) {}
__kernel void dctApproximate( double *image, double *result, int subblock) {}
int subblocks=2*4 , subblockSize=4*2 , blockSize=32 , imgW=1920 , imgH=1080 ;
double sgnflut [ ] = { 1 , . 9 , . 7 , . 3 , 7 . 8 , . 4 , . 3 , . 1 } ;

void DCT( double *image , double * result , double sgnfratio ) {
        for ( int id = 0 ; id < subblocks; id++) {
                #pragma acl task in ( image ) out (& result[ id * subblockSize] ) label("dct") \
                significant( sgnflut[id] ) approxfun( dctApprox ) workers( blockSize, blockSize ) \
                groups(imgW , imgH )
                dctAccurate ( image, result, i d ) ;
        }
        #pragma acl taskwait ratio( sgnfratio ) label ( "dct" )
}
```

Measuring energy/power

Access a set of hardware counters for measuring power and energy.

Intel's Running Average Power Limit (RAPL).

NVIDIA Management Library (NVML).

Implementation

Design decisions

Use OpenCL for code portability.

Reuse OpenCL implementations by Intel and Nvidia.

Asynchronous memory transfers and execution.

General Architecture

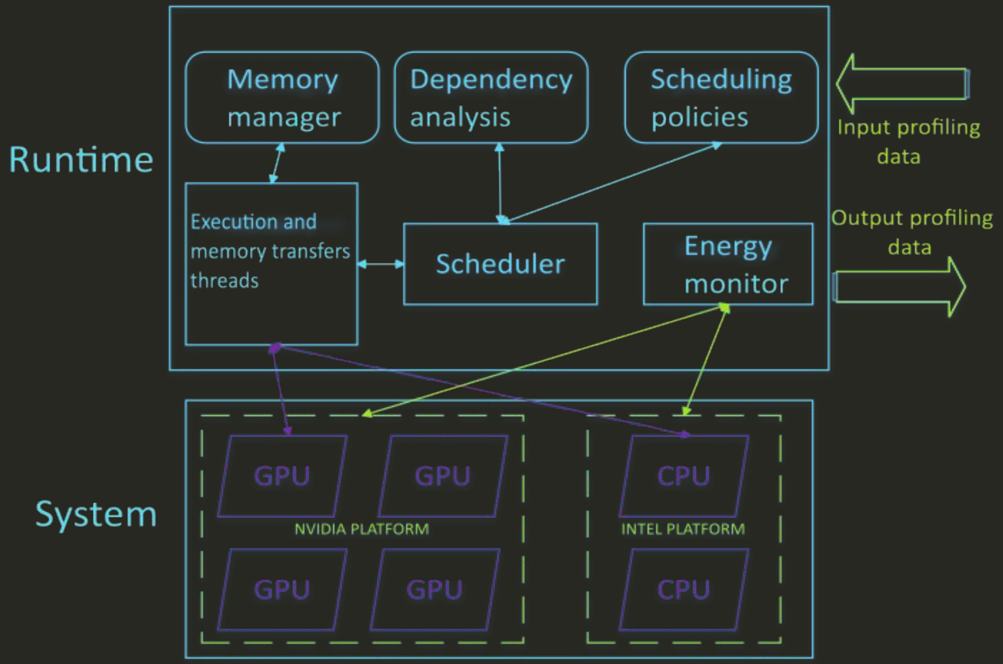
Collect and use profiling information.

Memory manager.

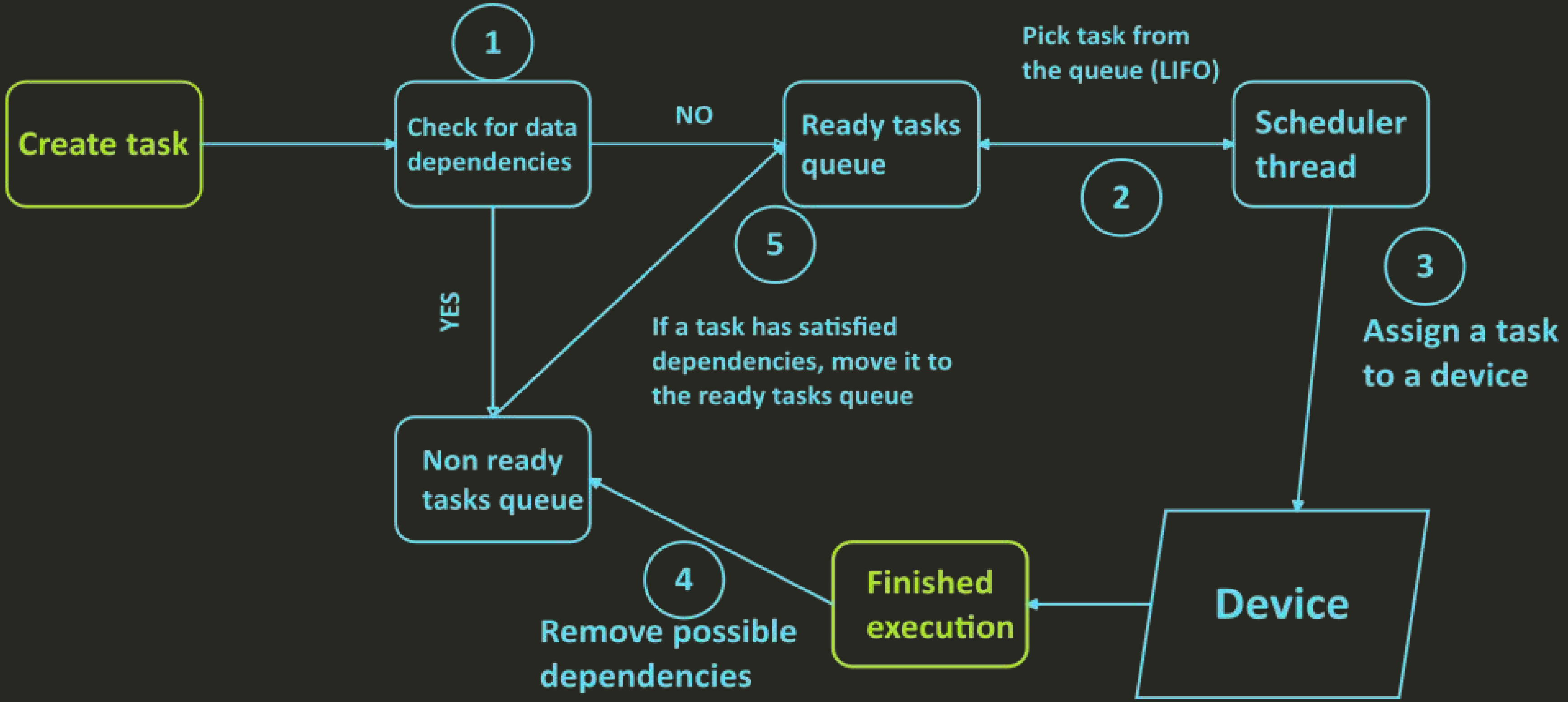
Dependence analysis between tasks.

Real time power/energy monitoring.

Scheduler and scheduling policies.



Life of a task



Data flow analysis

Applications often have dependencies:

WaW

```
#pragma acl task in(A) out(B)
#pragma acl task in(A) out(B)
```

WaR

```
#pragma acl task in(A) out(B)
#pragma acl task in(C) out(A)
```

RaW

```
#pragma acl task in(A) out(B)
#pragma acl task in(B) out(C)
```

Identify and enforce correct execution between them.

Data flow analysis example

```
#pragma acl task in(A) out(B)
task1(A,B)

#pragma acl task in(B) out(C)
task2(B,C)
```

Data flow analysis example

```
#pragma acl task in(A) out(B)
task1(A,B)

#pragma acl task in(B) out(C)
task2(B,C)
```

Hash table

key	value	
	in	out

Data flow analysis example

#pragma acl task in(A) out(B)
task1(A,B)

update

#pragma acl task in(B) out(C)
task2(B,C)

Hash table

key	value	
	in	out
&A	task1	NULL
&B	NULL	task1

Data flow analysis example

#pragma acl task in(A) out(B)
task1(A,B)

#pragma acl task in(B) out(C)
task2(B,C)

update

Hash table

key	value	
	in	out
&A	task1	NULL
&B	task2	task1
&C	NULL	task2

Data flow analysis example

```
#pragma acl task in(A) out(B)
task1(A,B)
```

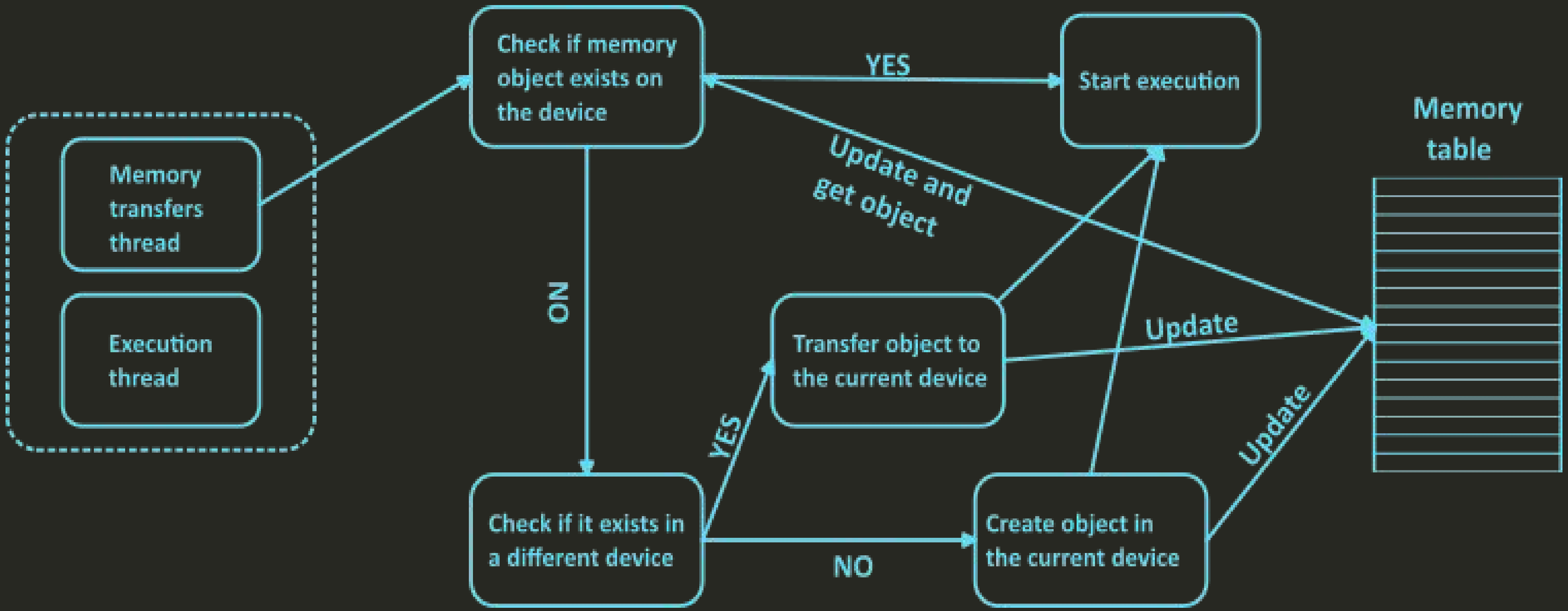
```
#pragma acl task in(B) out(C)
task2(B,C)
```

Hash table

key	value	
	in	out
&A	task1	NULL
&B	task2	task1
&C	NULL	task2

Dependency found for object B! task2 waits until task1 finishes execution.

Data management



Memory table

Store OpenCL objects (memory/kernels/programs).

For each object, store:

- State: Transferring, shared, exclusive or invalid.
- Owner: Device/s that has latest data.

Profiling support

Output data:

- task’s execution time
- data transfers time
- energy/power consumption

Input data:

- estimation functions for time and energy/power

Scheduling policies

Scheduling decisions

Execute application with profiling data:

Better time/energy estimation.

Without profiling data:

Runtime keeps history and tries to make an estimation.

Minimize execution time / energy consumption

Data locality and resource availability are the main criteria.

Policies are for the entire application.

Estimate execution time and energy consumption for each device.

Minimize time: try to distribute tasks across multiple devices.

Minimize energy: send task to the device with the smallest estimated energy consumption.

Energy budget policy

Targets

- Stay within budget
- Keep quality as high as possible
- Minimize execution time
- Only working per group

Energy budget policy

Targets

- Stay within budget
- Keep quality as high as possible
- Minimize execution time
- Only working per group

Algorithm

Step 1 → Estimate energy consumption for each $task_i$ in taskgoup and select a device for execution.

Energy budget policy

Targets

- Stay within budget
- Keep quality as high as possible
- Minimize execution time
- Only working per group

Algorithm

Step 1 → Estimate energy consumption for each $task_i$ in taskgoup and select a device for execution.

Step 2 → While estimated energy > budget, select the device with the lowest energy consumption.

Energy budget policy

Targets

- Stay within budget
- Keep quality as high as possible
- Minimize execution time
- Only working per group

Algorithm

Step 1 → Estimate energy consumption for each $task_i$ in taskgoup and select a device for execution.

Step 2 → While estimated energy > budget, select the device with the lowest energy consumption.

Step 3 → Start approximating tasks, starting with the least significant ones.

Energy budget policy

Targets

- Stay within budget
- Keep quality as high as possible
- Minimize execution time
- Only working per group

Algorithm

Step 1 → Estimate energy consumption for each $task_i$ in taskgroup and select a device for execution.

Step 2 → While estimated energy > budget, select the device with the lowest energy consumption.

Step 3 → Start approximating tasks, starting with the least significant ones.

Step 4 → Drop entire tasks in order to meet energy budget.

Experimental Evaluation

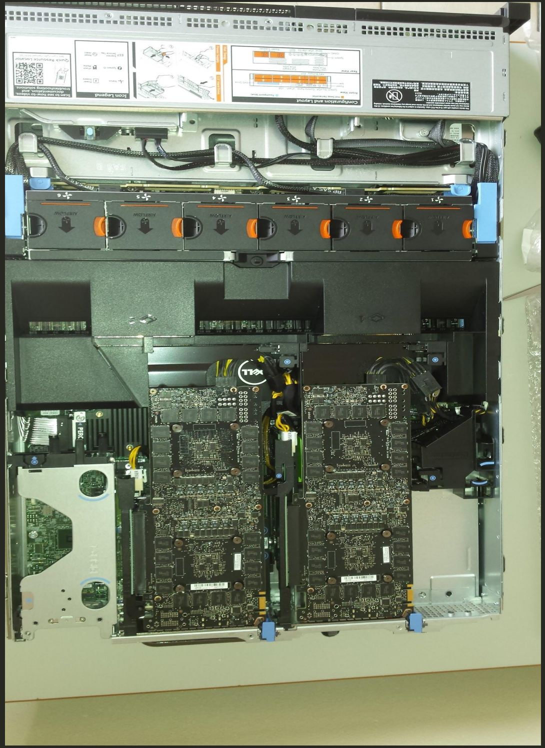
System configuration

Two Intel Xeon E5 2695 @ 2.3Ghz, 14 cores each

Two Nvidia Tesla K80

128Gb RAM

Power/energy sampling every 2ms



Applications

HOG → Computer vision, pedestrian detection

CG → Dense algebra

BONDS → Financial

SPStereo Disparity → Computer vision, depth estimation

PBPI → Bioinformatics, phylogenetic

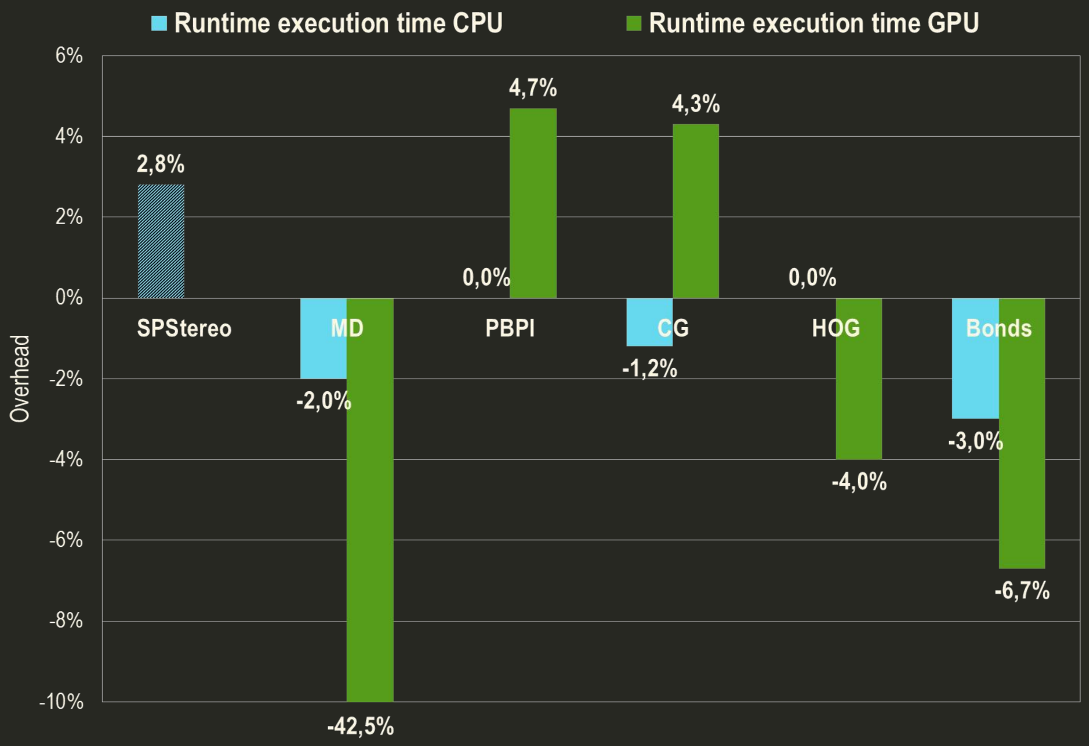
MD → Molecular Dynamics

Runtime Overhead

Pure OpenCL vs runtime, using 1 device only (CPU or GPU).

Low overhead in most cases.

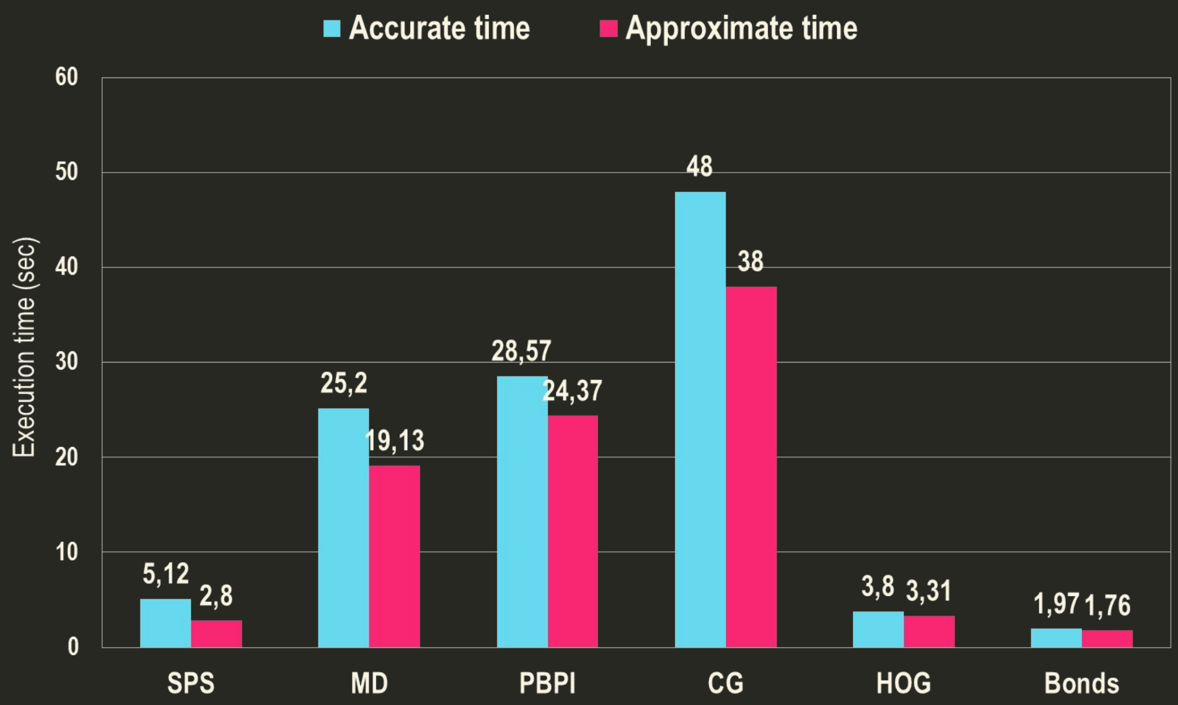
Some applications benefit from multiple command queues.



Approximation gains - Time

Distribute tasks amongst devices.

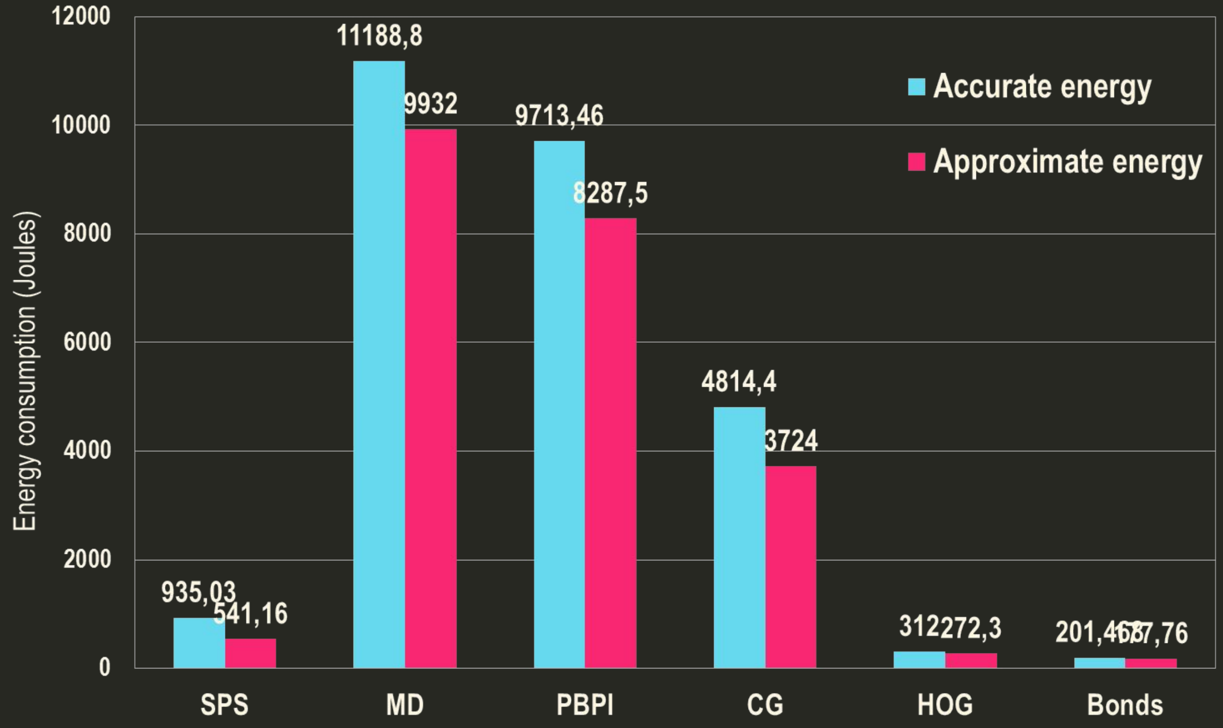
Quality stays in acceptable rates.



Approximation gains - Energy

Significant energy gains in all applications.

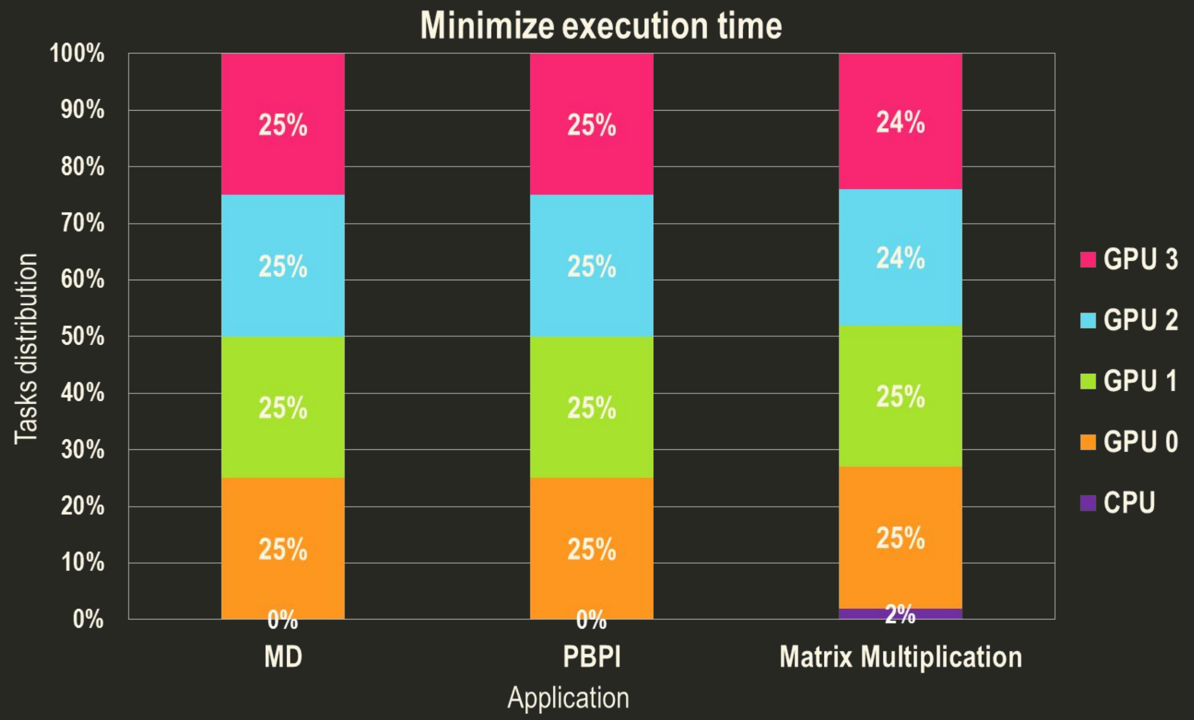
SPS 42.1%
MD 11.2%
PBPI 14.6%
CG 22.6%
HOG 12.8%
Bonds 11.9%



Minimize execution time - Tasks distribution

Having profiling data,
makes the decision easier.

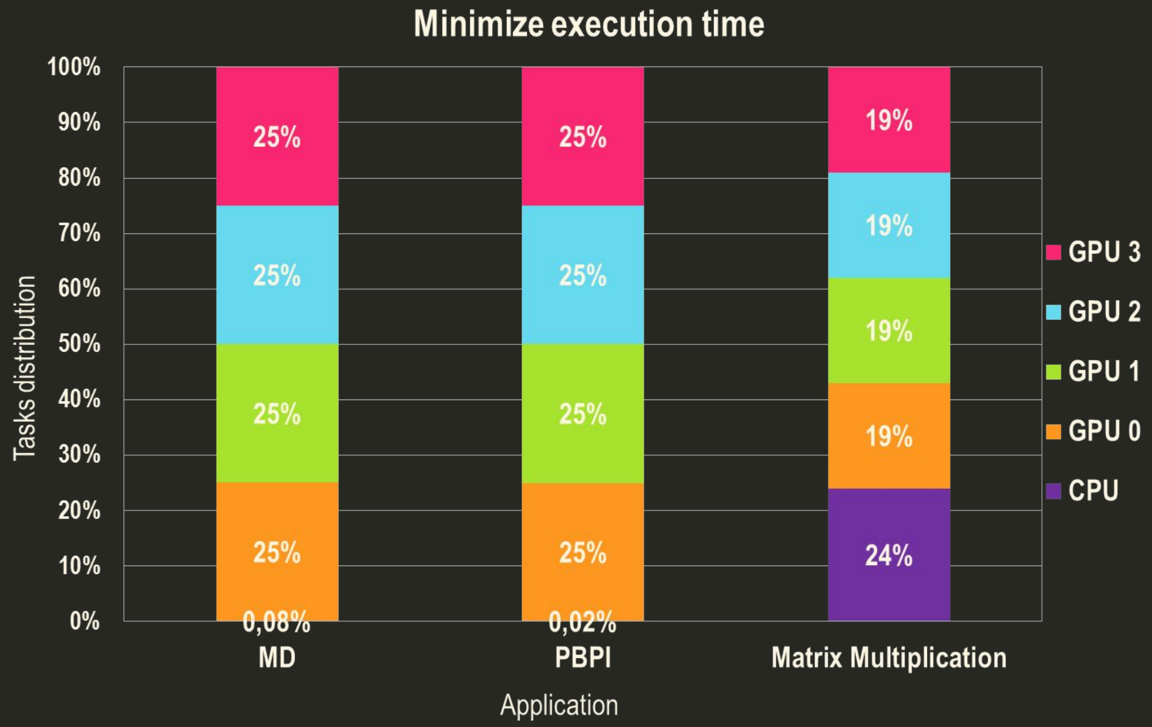
MD → 4,000 tasks
PBPI → 10,000 tasks
MM → 100 tasks



Minimize execution time - Tasks distribution

Even without profiling data, runtime's performance is similar.

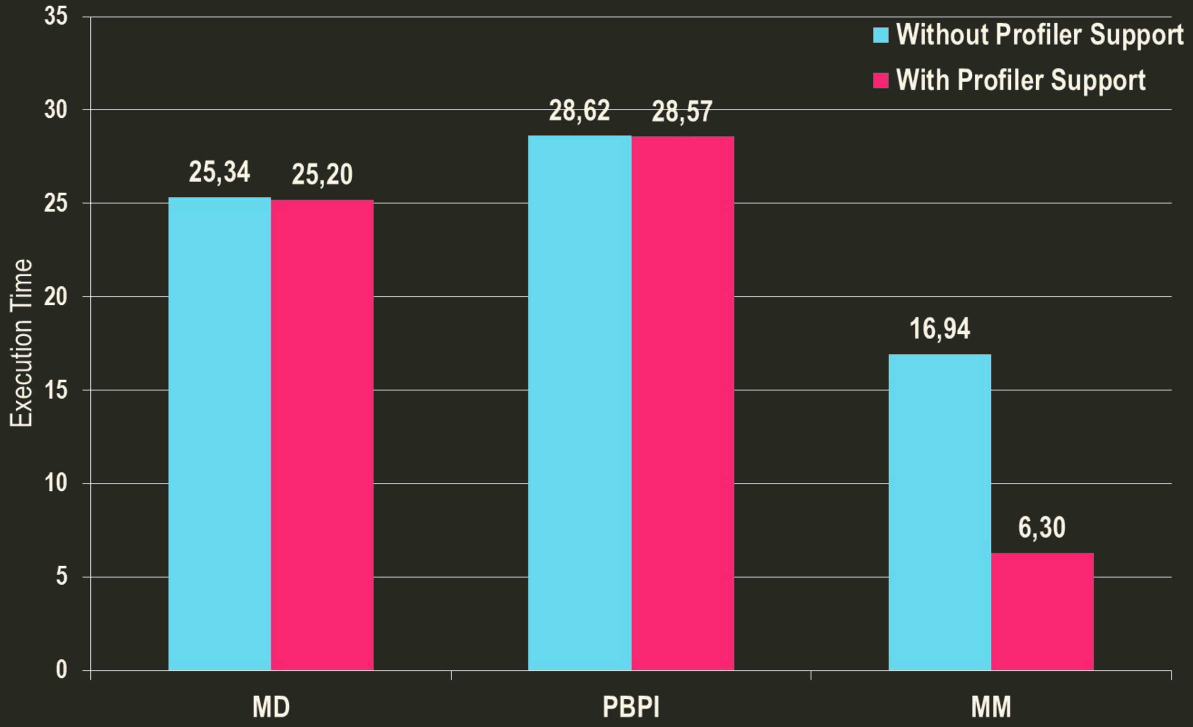
MM spawns 100 tasks instantly.



Minimize execution time - With & without profiler

Similar behaviour for MD and PBPI.

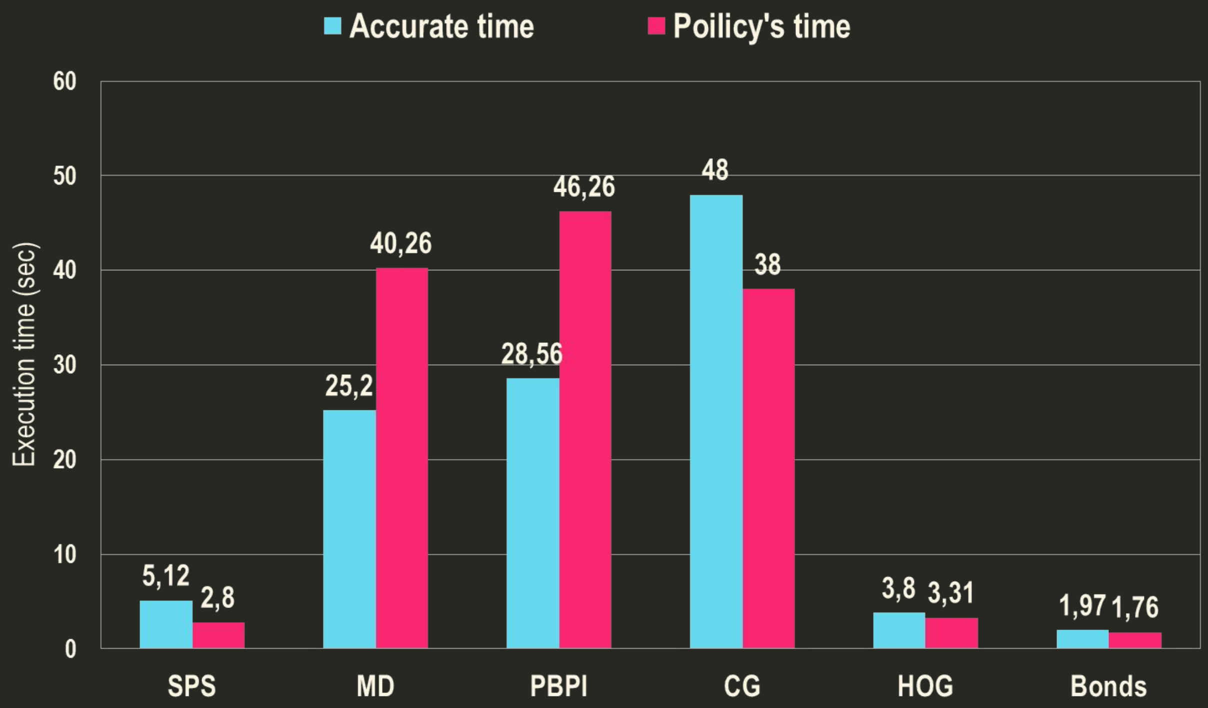
Bad case: Matrix Multiplication



Minimize energy consumption - Time

Execution time is much higher.

Tasks are executed approximately.

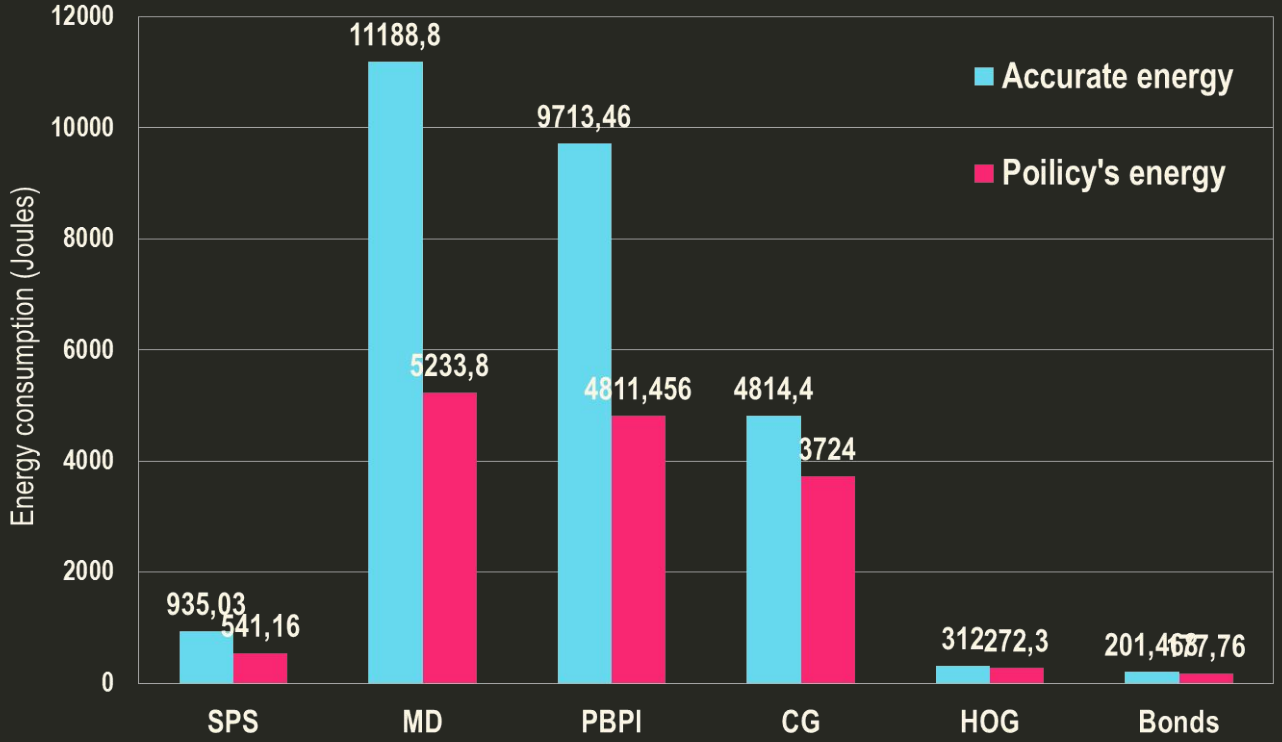


Minimize energy consumption - Energy

..but great energy gains!

MD 53.2%

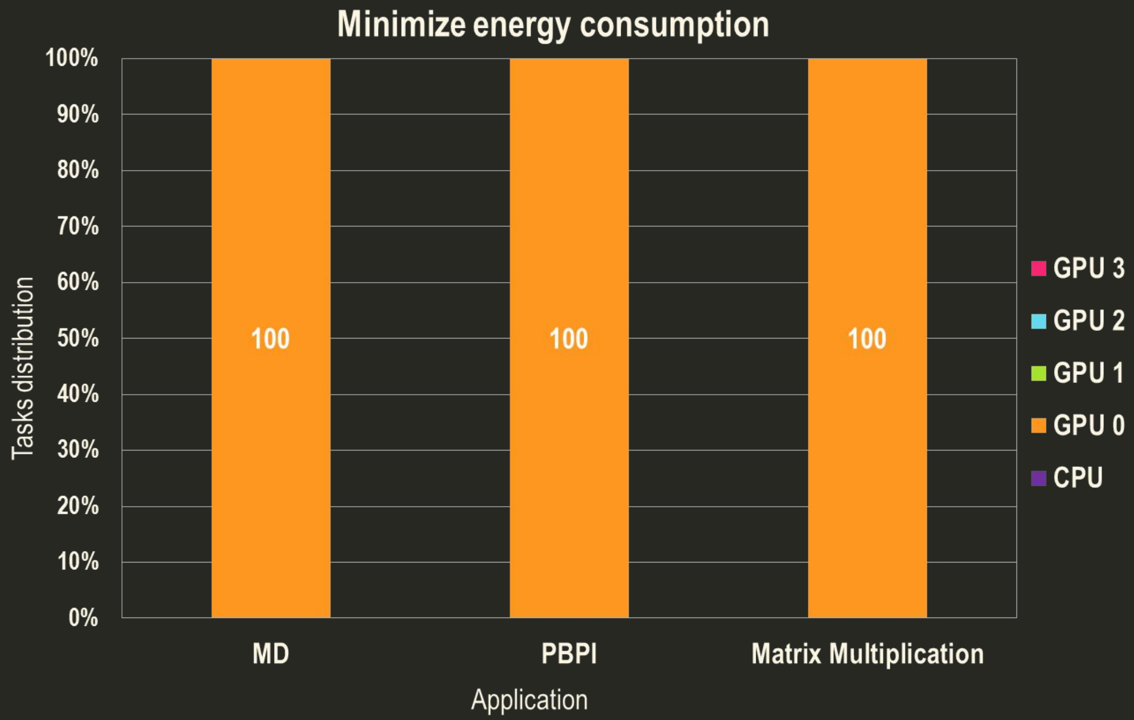
PBPI 50.4%



Minimize energy consumption - Tasks distribution

Executing on 1 device (GPU) is the most energy efficient configuration.

Profiling data help the runtime figure this out.

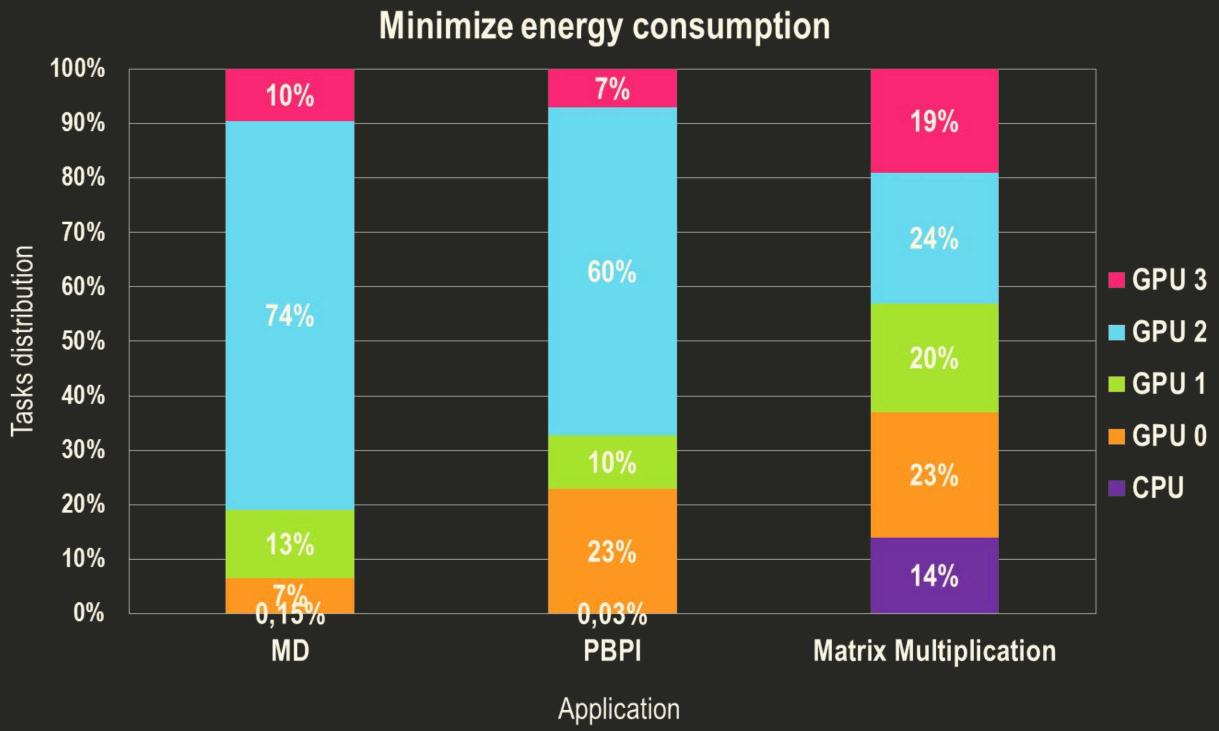


Minimize energy consumption - Tasks distribution

Without profiling data, runtime tries different configurations.

Takes time to find the optimal one.

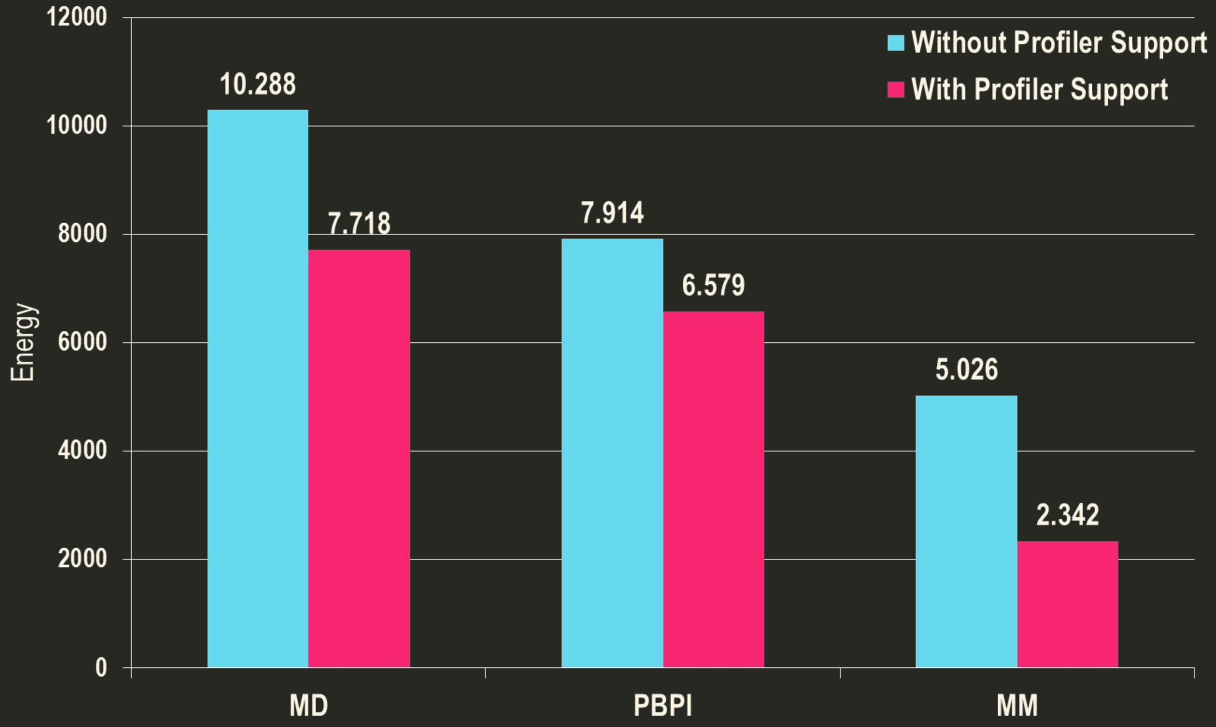
Real time power readings make estimation harder.



Minimize energy consumption - With & without profiler

Searching for the optimal configuration has a negative impact.

MD: 24,9%
PBPI: 16,8%
MM: 53,4%



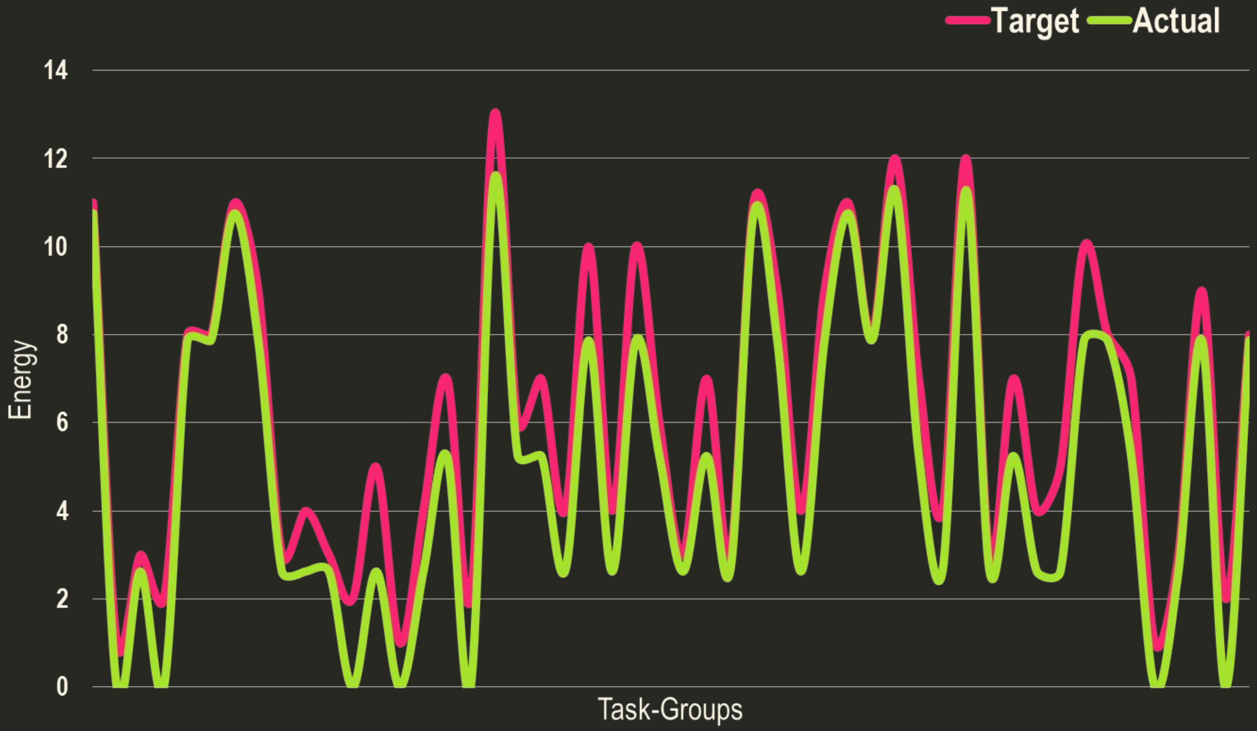
Energy budget policy - Runtime adaption

Budget limits are hard.

Always below budget.

For some cases this is extreme.

What if we used the energy that we saved from previous iterations?



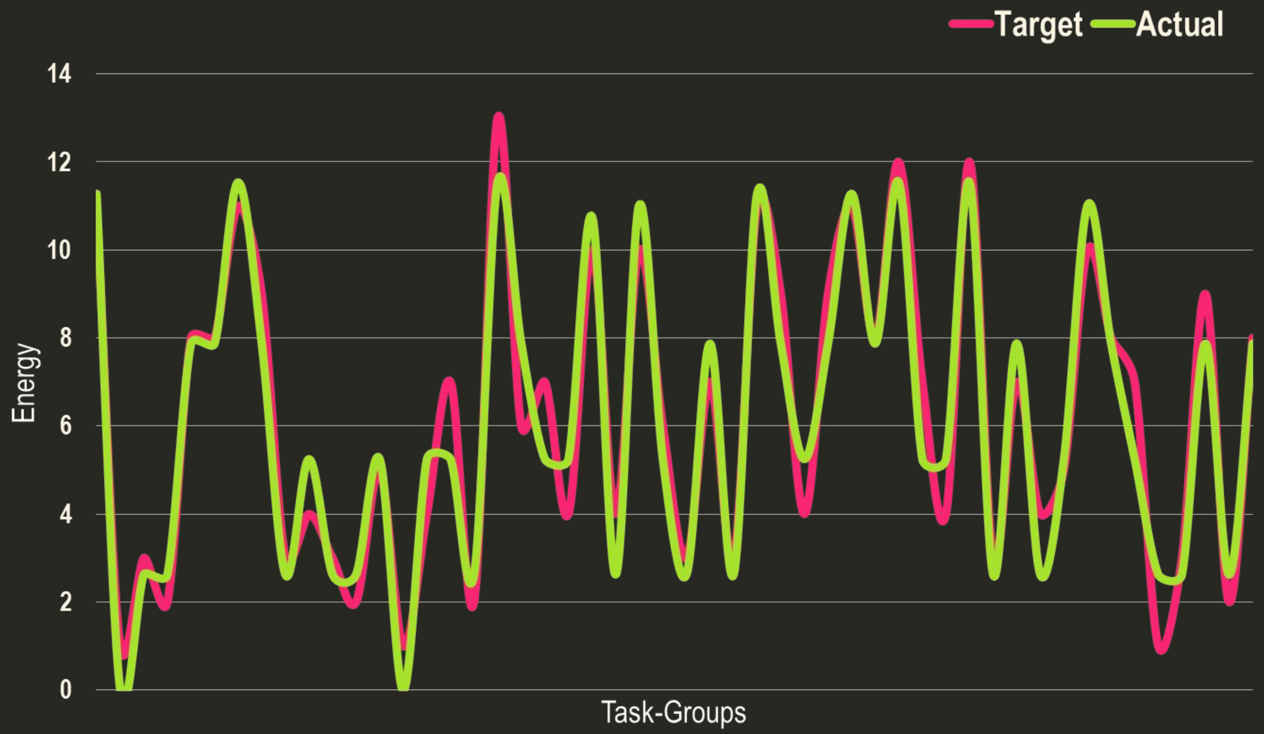
Energy budget policy - Runtime adaption v2

What we gained.

Cumulative energy budget: 6737 J

v1: 5541.58 J
AC/AP/D: 531/1533/1936

v2: 6632.01 J
AC/AP/D: 827/1619/1554

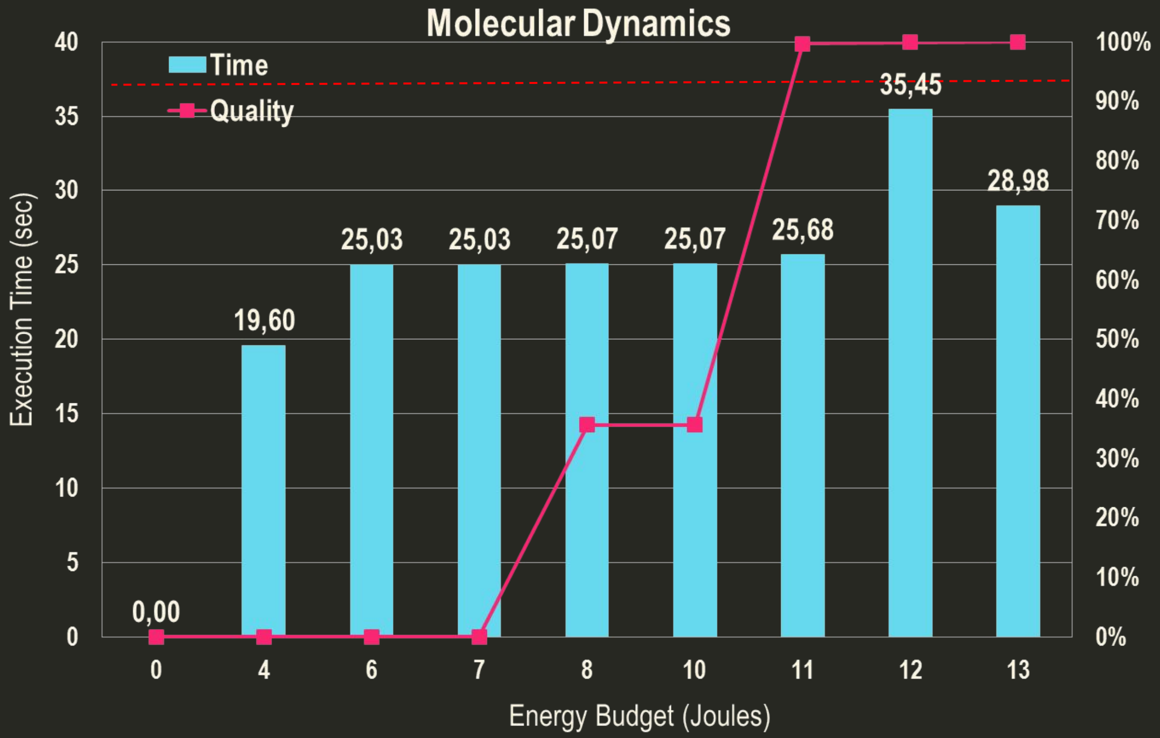


Energy budget policy

4 tasks need ~12.41 Joules

At 11 Joules, 3,000 tasks approximate, 1,000 accurate.

Below 10 Joules, quality is not accepted.

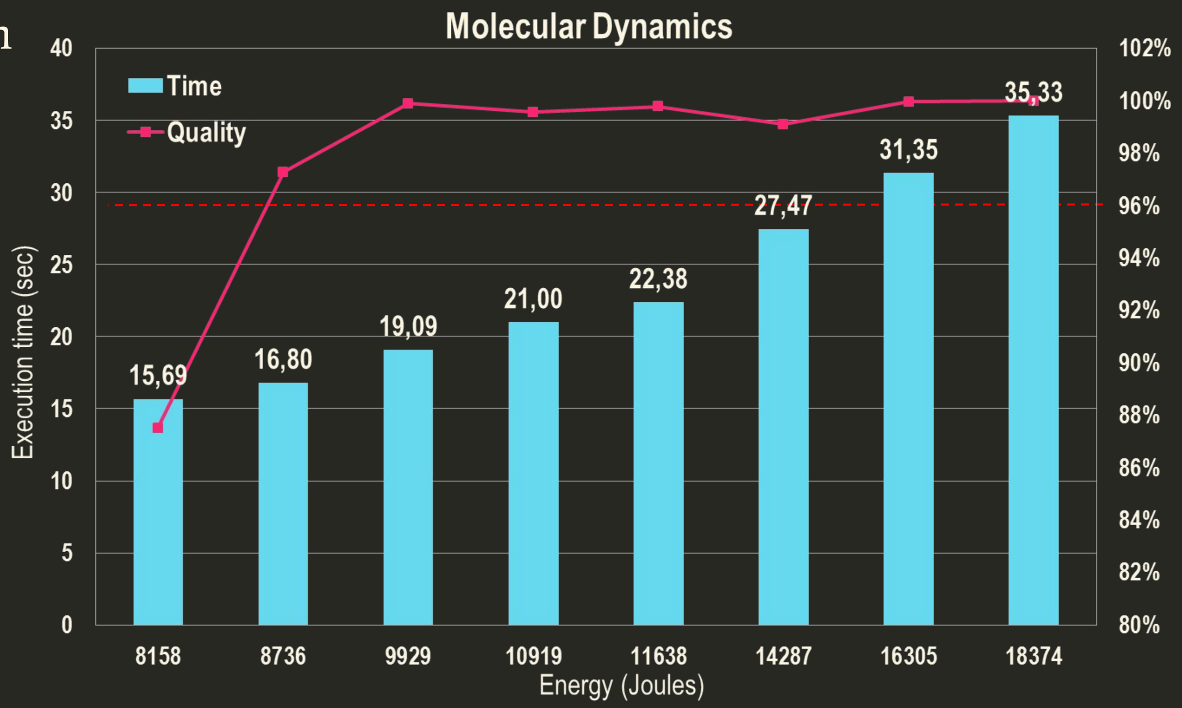


Energy budget policy

Try different budgets in each iteration.

For example, first taskgroup is executed with a budget of 14J, second with 10J etc.

Due to the nature of MD, we get good results even for low energy.



Conclusion

Some stats

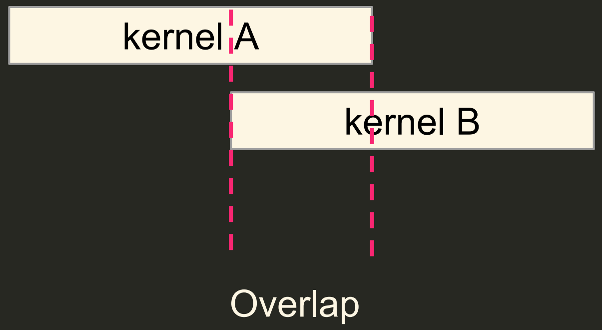
LOC: 6214
Files: 38

Q & A

Backup slides

Estimate execution time and energy consumption

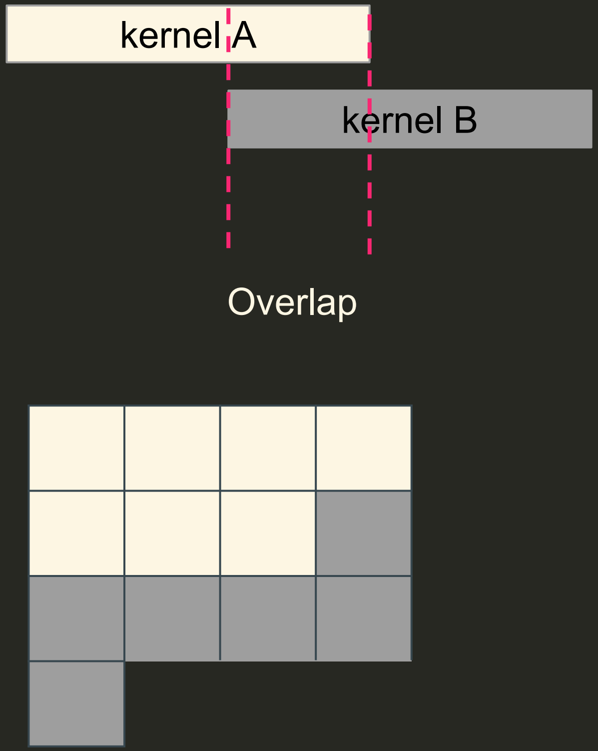
Overlapping kernels must be taken into account.



Estimate execution time and energy consumption

Overlapping kernels must be taken into account.

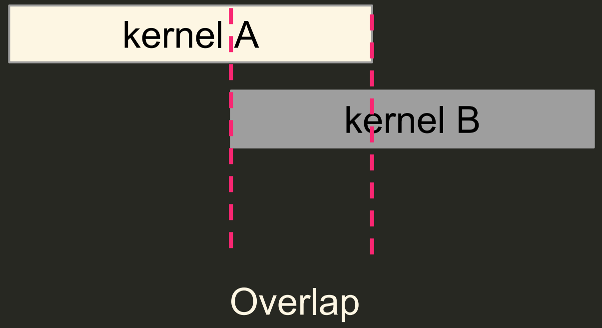
Kernels A and B have 20 blocks each, 13 SMs in our GPU.



Estimate execution time and energy consumption

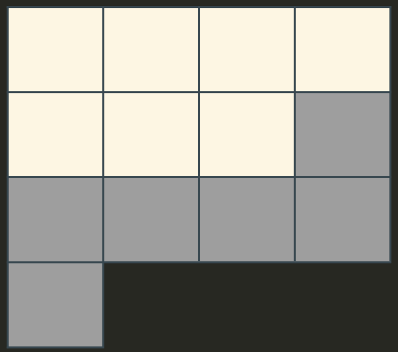
Overlapping kernels must be taken into account.

Kernels A and B have 20 blocks each, 13 SMs in our GPU.



We can calculate approximately the overlap time:

$$Overlap\_time(A) = pred\_time(A) \times \frac{blocks(A) mod SM}{blocks(A)}$$



Centaurus programming model

```
#pragma acl task [approxfun( function )]
                 [significant( expr )]
                 [in( varlist )] [out( varlist )] [inout( varlist ) ]
                 [device_in( varlist )] [device_out( varlist )]  [device_inout( varlist )]
                 [workers( int_expr_list )] [groups( int_expr_list )]
                 [bind( device_type )]
                 [label( "name" )]
accurate_task_impl(…) ;

#pragma acl taskgroup label( string_expr )  [energy_joule( uint ) | ratio( double )]

#pragma acl taskwait [label( "name" )]
```

Some problems

Each callback function from NVIDIA adds ~20ms.

- For 1,000 tasks, ~60 sec overhead.
- Solution: Polling thread.

Overlapping kernels:

- OpenCL events on multiple command queues, return wrong timestamps.
- Solution: Try to estimate overlap time.