Optimization of a MATLAB Iterative Kernel

Performance Comparisons Between GPGPU and the Polyhedral Model

a project by
Edoardo Mondoni

tutored by
Riccardo Cattaneo

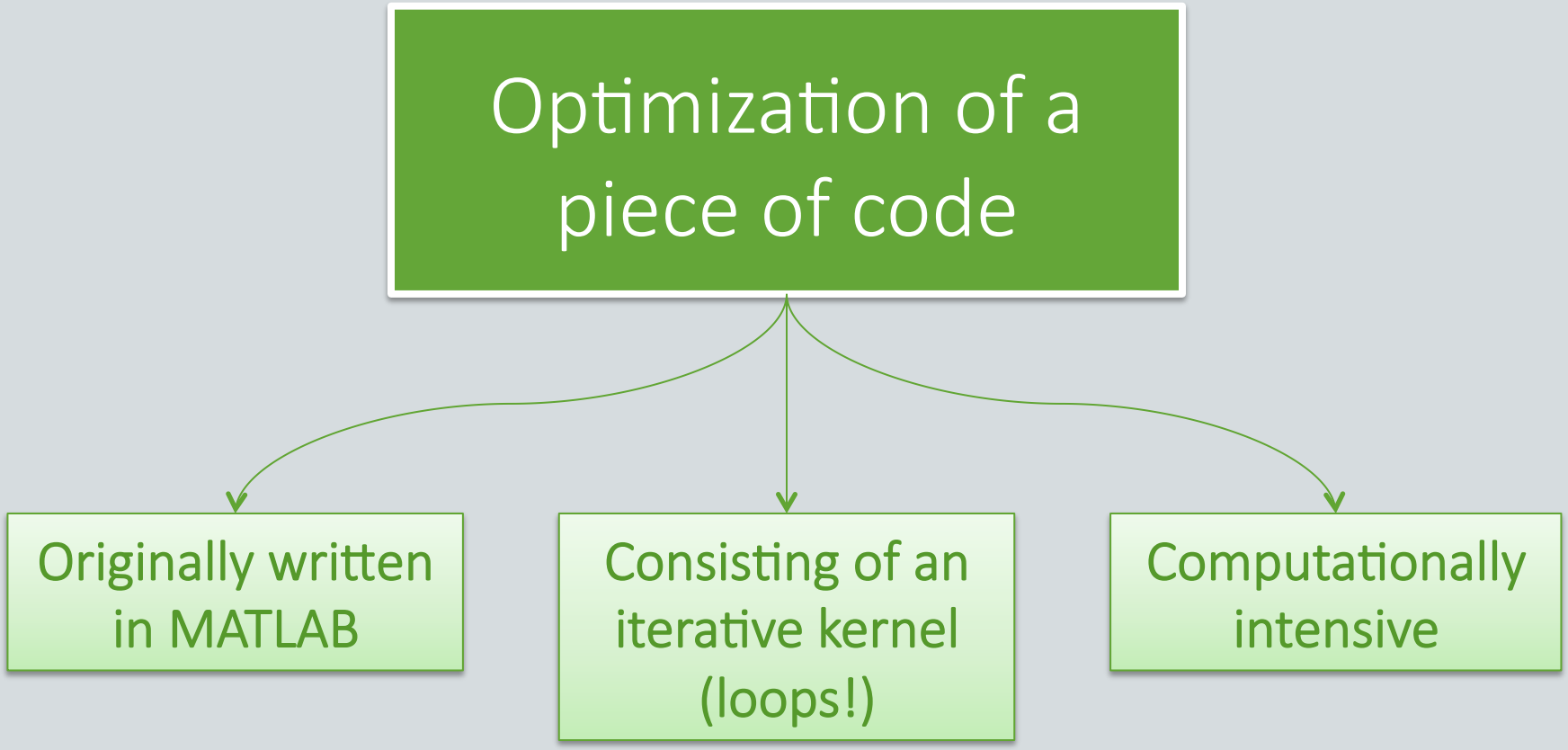
High Performance Processors and Systems – Research Project
Prof. Donatella Sciuto and Marco D. Santambrogio
a.y. 2013/14

Outline

	a.k.a.
• Problem description	*What had to be done*
• State of the art	*How could it be done*
• Project summary	*What was actually done*
• Results and conclusions	*How it turned out*
• Further developments	*What else can be done*

Problem Description

Our Goal


Optimization of a piece of code
Originally written in MATLAB
Consisting of an iterative kernel (loops!)
Computationally intensive

The Code: `coupled_oscillators`

- Simulates the behaviour of an array of interconnected LC oscillators over time
- Simulation parameters:
  - n_{osc} (i.e. the **size of the system**)
  - n_{steps} (i.e. the simulation **time frame**)
- For each time step t_i and for each oscillator k, computes: $\alpha_k(t_i)$ and $\theta_k(t_i)$

Load samples
- Experimental data detailing the oscillators' features

Generate Υ matrix
- Contains mutual interference weights
- At random

Initialize simulation
- Determine simulation parameters
- Allocate memory for outputs and randomize $\alpha_k(0)$

Simulate
- Calculate $\alpha_k(t)$ and $\theta_k(t)$, for each t_i and each k

Computational Complexity

The **`simulate`** kernel

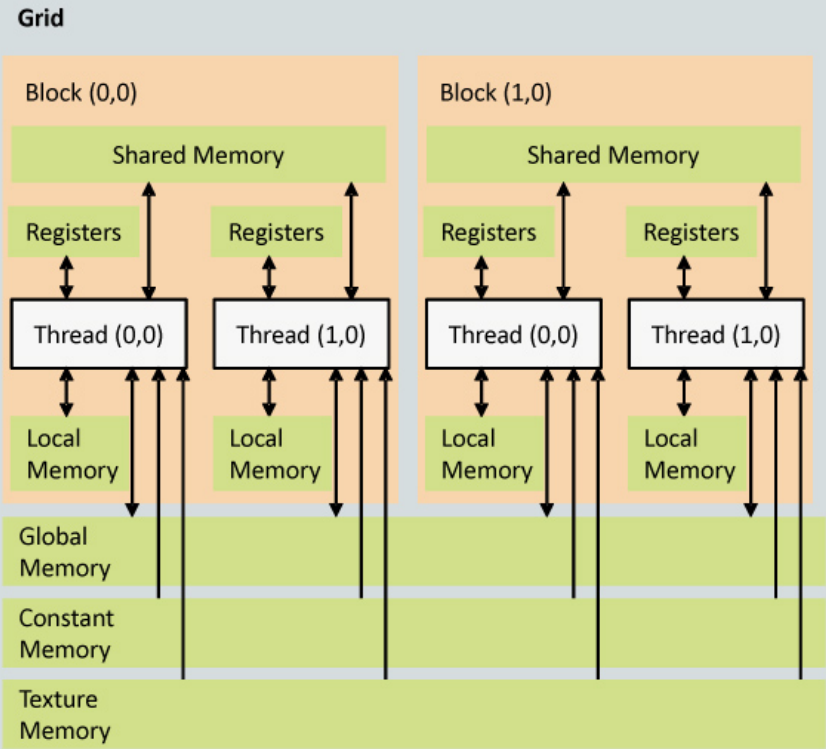
```
unsigned int t, k, kk;

for (t = 1; t < n_steps; t++) {
  for (k = 0; k < n_osc; k++) {
    for (kk = 0; kk < n_osc; kk++) {
      //determine oscillator kk's
      //interference on oscillator k
      //using Y's values
    }

    //calculate & store alpha_k(t)
    //calculate & store theta_k(t)
  }
}
```

- `simulate`'s complexity amounts to $O(n_{\text{steps}} \cdot n_{\text{osc}}^2)$
  - linear wrt the simulation time frame
  - quadratic wrt to the size of the system
- In our tests: 1k ÷ 10k oscillators, 25k steps
  - > 10^{11} inner loop iterations!

State of the Art

CUDA

- Clear-cut distinction
  - Device (i.e. the GPU): executes kernels
  - Host (i.e. the CPU): executes everything else as in ordinary programs
- Implements a Single Instruction Multiple *Thread* (SIMT) paradigm
- Very stratified memory hierarchy
  - Where to place data?
    - Depends on data size…
    - … on data access patterns…
    - … and on data visibility needs

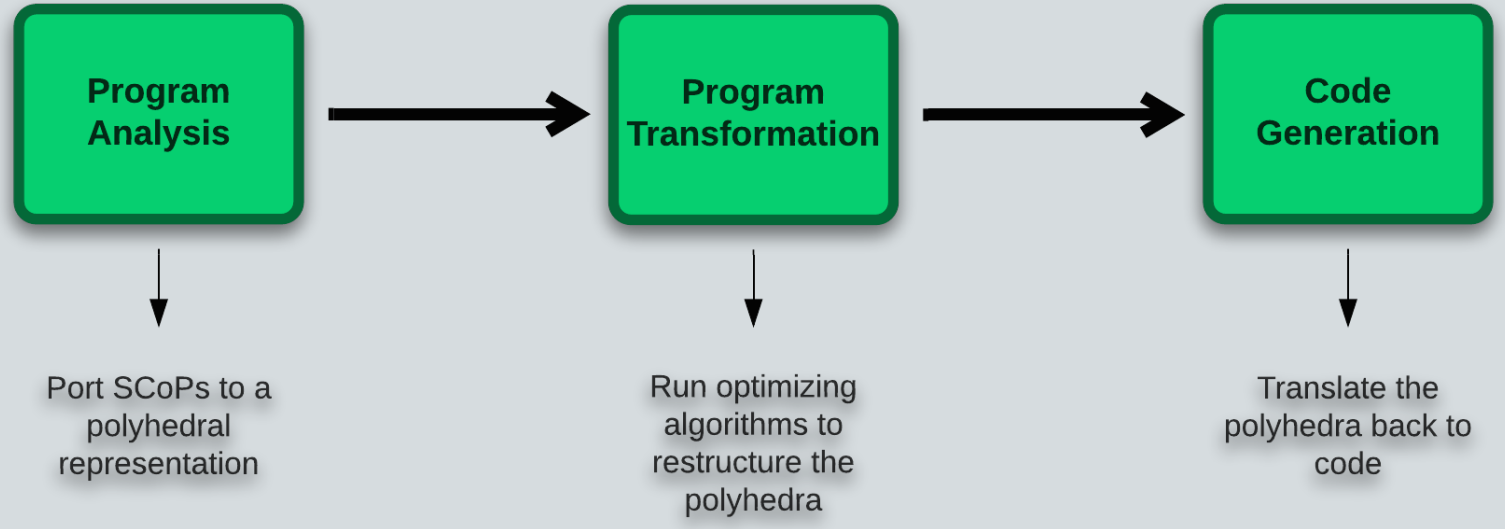


Choosing the right one is **vital to performance!**

The Polyhedral Framework

- Mathematical framework
- Maps the iteration domain of a statement enclosed in n nested loops to an integer polyhedron in n dimensions

- Why even bother?
  - Easier to manipulate loops through algebraic transformations
  - Much easier to keep data dependencies into account



Project Summary

What It Consisted Of

- I rewrote the program from scratch. *Twice.*
- **Plain C** version: static optimizations
  - `-O3` compilation
  - PLuTO: polyhedral optimizer
- **CUDA C/C++** version: progressive refinements
  - Global memory coalescing
  - Use of constant memory
  - Use of shared memory

Plain C Version

- `-O0` is the default compilation mode
  - No optimization whatsoever
- `-O3` turns on nearly all `-fx` optimization flags
  - Increases compilation time (or does it?)
  - Boosts program's performance
- *PLuTO*: source-to-source polyhedral optimizer
  - Parses ordinary `.c` files
  - Outputs reworked `.c` files, based on desired optimizations
  - `--tile`, `--parallel`
    - Loop tiling = loop blocking
    - Loop parallelization through the OpenMP specifications

PLuTO in Action

Original source code

```
unsigned int t, k, kk;

for (t = 1; t < n_steps; t++) {
  for (k = 0; k < n_osc; k++) {
    current = 0.0;

    for (kk = 0; kk < n_osc; kk++)
      current += //math;

    alpha_cur[k] = //more math;
    theta_cur[k] = //still math;
  }

  //pointer black magic follows
}
```

$O(n_{steps} \cdot n_{osc}^2)$
Untouched

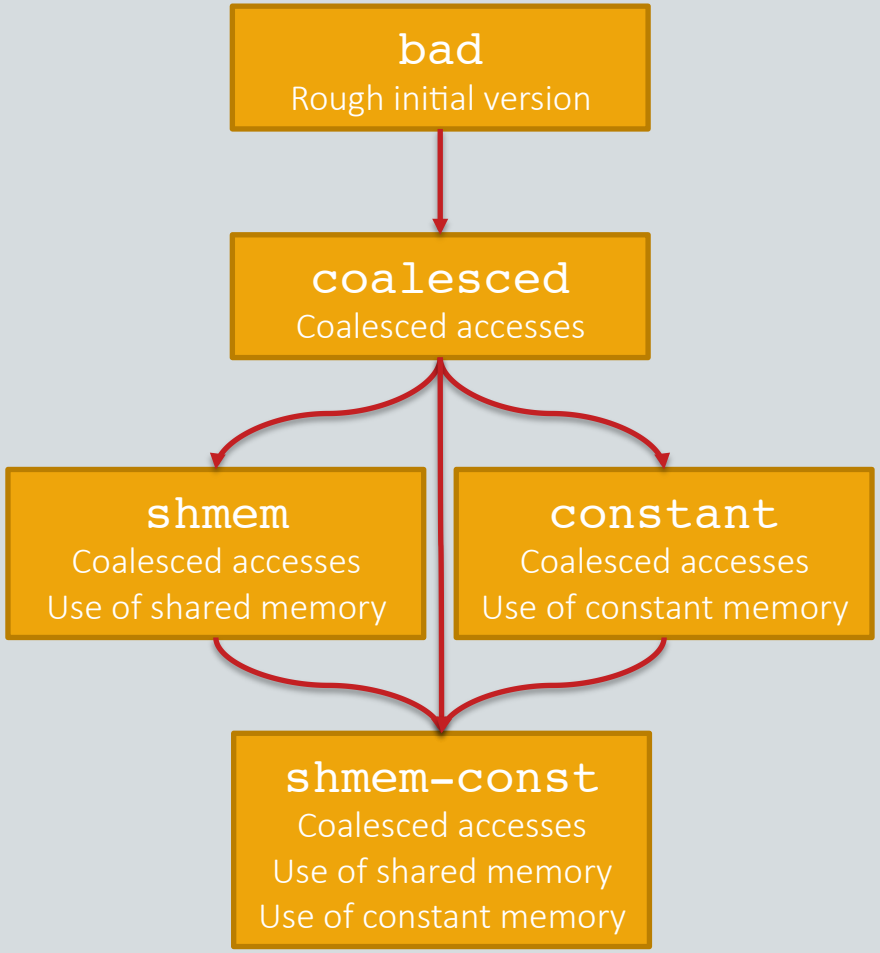
Optimized code

```
for (t2=1;t2<=n_steps-1;t2++) {
  for (t4=0;t4<=n_osc-1;t4++) {
    current = 0.0;

    for (t6=0;t6<=n_osc-1;t6++)
        current += //math;

    alpha_cur[t4] = //more math;
  }

  lbp=0; ubp=n_osc-1;
  #pragma omp parallel for
   private(lbv,ubv,t5,t6)
  for (t4=lbp;t4<=ubp;t4++)
    theta_cur[t4]= //still math;

  //pointer black magic follows
}
```

$O(n_{steps} \cdot n_{osc})$
Parallelized

CUDA C/C++ Version

- Developed an initial version
  - "Whatever works"
  - Roughly the same as plain C program, save for kernels and `cudaMemcpy` calls
- Then, refined it through *memory optimizations only*
  - Memory access patterns…
  - … and smart choice of memory positioning for different types of data



Memory Access Coalescing

Strided accesses

```
for (kk = 0; kk < d_n_osc; kk++)
  current += matrix[kk + idx * d_n_osc] …;
```

idx\kk	0	1	2	3	4
0	T_0, t_0	T_0, t_1	T_0, t_2	T_0, t_3	T_0, t_4
1	T_1, t_0	T_1, t_1	T_1, t_2	T_1, t_3	T_1, t_4
2	T_2, t_0	T_2, t_1	T_2, t_2	T_2, t_3	T_2, t_4
3	T_3, t_0	T_3, t_1	T_3, t_2	T_3, t_3	T_3, t_4
4	T_4, t_0	T_4, t_1	T_4, t_2	T_4, t_3	T_4, t_4

- At any given time t_i, concurrent threads $T_0 \ldots T_4$ access very distant memory locations
- No spatial locality!

Coalesced accesses

```
for (kk = 0; kk < d_n_osc; kk++)
  current += matrix[idx + kk * d_n_osc] …;
```

kk\idx	0	1	2	3	4
0	T_0, t_0	T_1, t_0	T_2, t_0	T_3, t_0	T_4, t_0
1	T_0, t_1	T_1, t_1	T_2, t_1	T_3, t_1	T_4, t_1
2	T_0, t_2	T_1, t_2	T_2, t_2	T_3, t_2	T_4, t_2
3	T_0, t_3	T_1, t_3	T_2, t_3	T_3, t_3	T_4, t_3
4	T_0, t_4	T_1, t_4	T_2, t_4	T_3, t_4	T_4, t_4

- At any given time t_i, concurrent threads $T_0 \ldots T_4$ access adjacent memory locations
- Accomplishes data locality!

Results and Conclusions

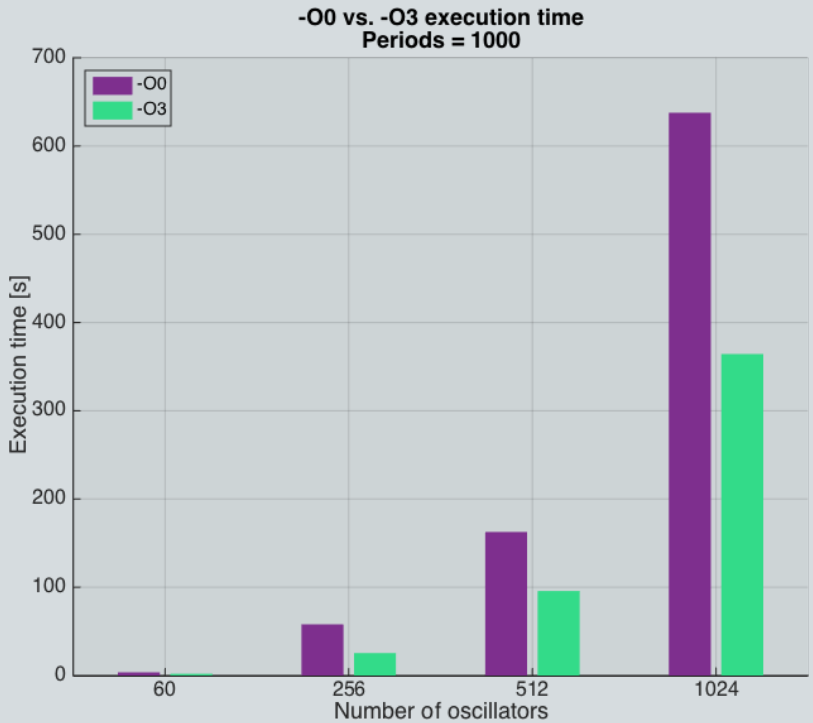
Experiment Outline

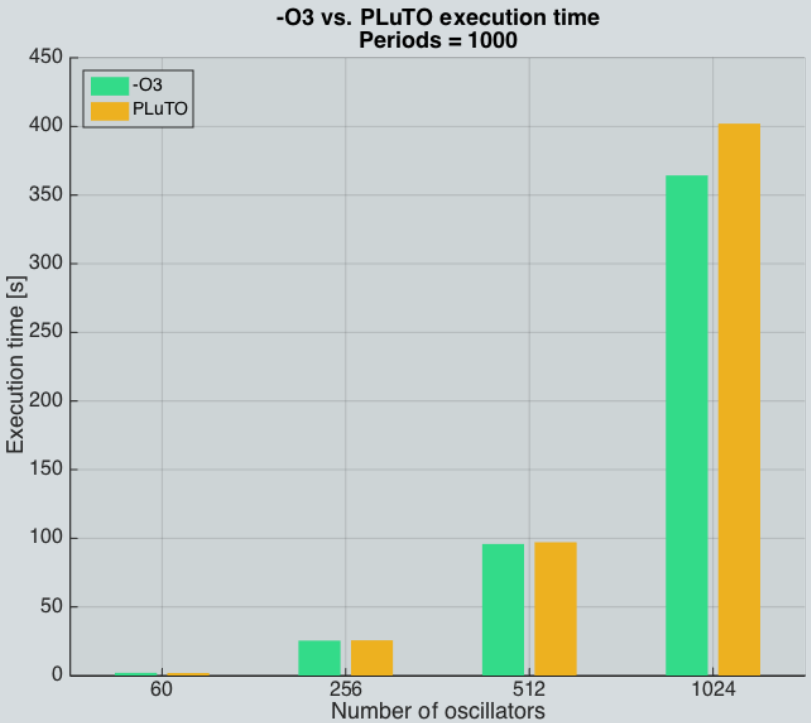
- The binaries were ran several times
- Each time with different (n_{osc}, n_{per}) parameter combinations
  - n_{osc} = 60, 256, 512, 1024, 5120, 10240
  - n_{per} = 100, 1000 (1 period = 25 steps)
- For each run, t_{ex} was measured

The comparisons

Plain C Version
`-O0` vs. `-O3`
`-O3` vs. PLuTO
CUDA C/C++ Version
`bad` vs. `coalesced`
`bad` vs. `constant`
`bad` vs. `shmem`
`bad` vs. `shmem-const`
Wrap-up
MATLAB vs. Plain C vs. CUDA C/C++

t_{ex}: Plain C Static Optimizations

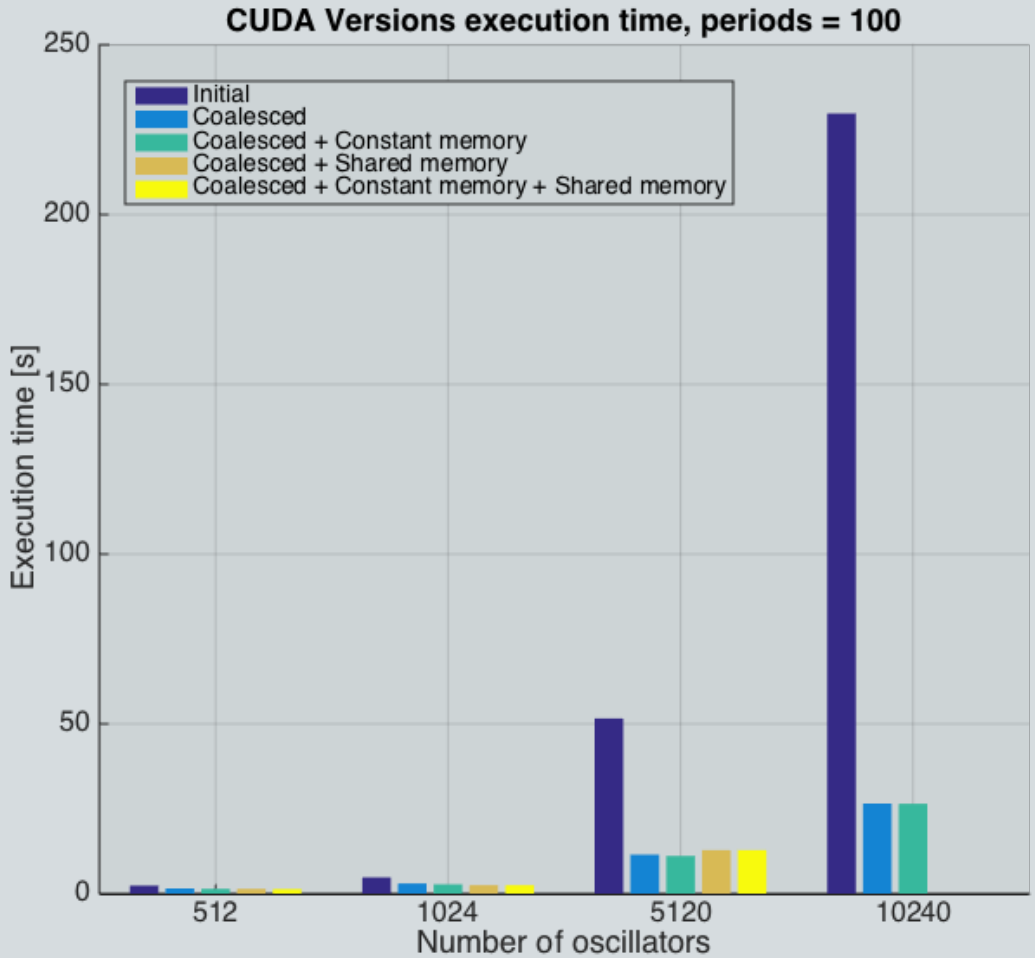




- Average speedup: 1.77x
- Very effective!

- Average speedup: 0.93x
- Tiling and parallelization overhead!

t_{ex}: CUDA Memory Optimizations

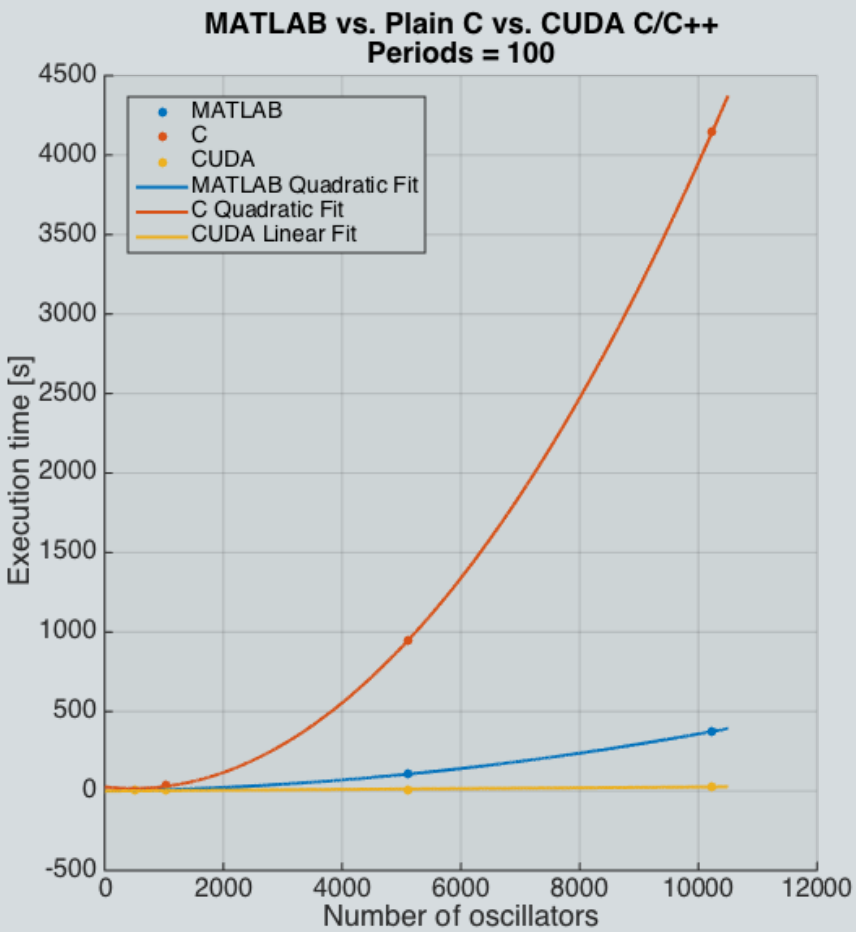


Average speedups	
`coalesced` vs. `bad`	6.77x
`const` vs. `bad`	6.92x
`shmem` vs. `bad`	3.51x
`shmem-const` vs. `bad`	3.52x

NOTE: `shmem` and `shmem-const` could not be tested for n_{osc} > 6144

- Impressive speedups!
  - 86% less time on average
- Coalescing alone accounts for most of the speedup

t_{ex}: MATLAB vs. Plain C vs. CUDA

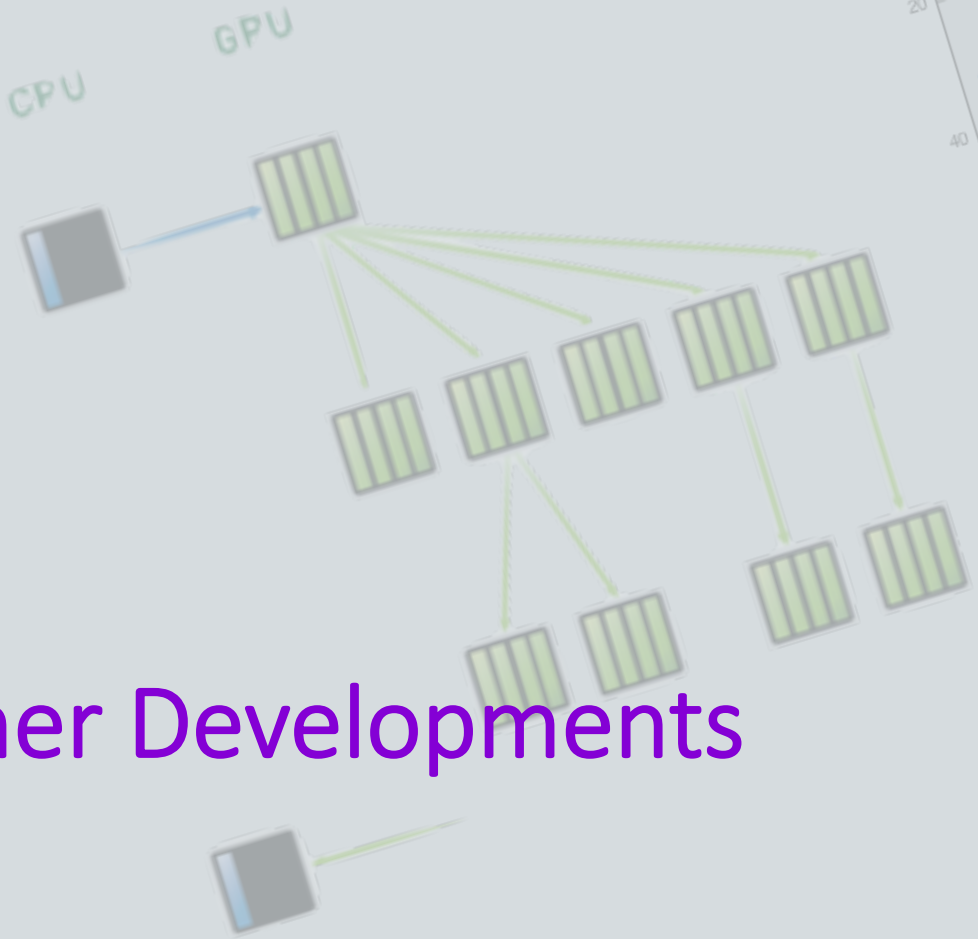


Speedups	*Min*	*Avg*	*Max*
C vs. MATLAB	0.09x	0.11x	1.70x
CUDA vs. MATLAB	1.66x	7.25x	14.2x

- Plain C way worse than MATLAB
  - Automatic vectorization in MATLAB?
- CUDA strikingly outperforms MATLAB
  - Speedup increases w/ n_{osc}
  - Downgrades computational complexity from quadratic to linear (wrt to n_{osc})

Conclusions

- -O3 is quite effective on C code
- PLuTO might fail to optimize code
  - Strict definition of SCoP
  - Heuristics
- Smart memory management is vital in CUDA
  - Coalescing memory accesses
  - Making wise use of the memory hierarchy
- Performance gap widens as n_{osc} grows
  - Because MATLAB's and C's t_{ex} grow quadratically…
  - … while CUDA's grows linearly



Further Developments

Beyond Memory Optimizations

- Diminishing returns on memory optimizations
  - The bottleneck now lies somewhere else
- How to improve performance even more?
  - CUDA Dynamic Parallelism
- How to get rid of existing constraints on program execution (memory footprint…)?
  - Implementation of cuSPARSE
  - Concurrent copy and kernel execution

CUDA Dynamic Parallelism

- Originally: kernels could not launch other kernels
- Then: Nvidia introduced Dynamic Parallelism (2012)
  - Running threads can call other kernels = spawn other threads
  - Nested grids
  - Requires Compute Capability 3.5 or greater
- How can it be useful?
  - Parallelize inner loop!
  - Further reduce algorithm complexity

```
unsigned int t, k, kk;

for (t = 1; t < n_steps; t++) {
  for (k = 0; k < n_osc; k++) {
    for (kk = 0; kk < n_osc; kk++) {
      //determine oscillator kk's
      //interference on oscillator k
```

SIMT completely flattens the middle loop

$$O(n_{steps} \cdot n_{osc}^2) \rightarrow O(n_{steps} \cdot n_{osc})$$

Dynamic Parallelism parallelizes the reduce operation in the inner loop

$$O(n_{steps} \cdot n_{osc}) \rightarrow O(n_{steps} \cdot \log_2 n_{osc})$$

Question Time!