EPSRC Centre for Doctoral Training in
Pervasive Parallelism

# Dynamic Autotuning of Algorithmic Skeletons

Chris Cummins

THE UNIVERSITY of EDINBURGH

informatics

EPSRC
Pioneering research and skills

# Objectives

Demonstrate **dynamic autotuning** of algorithmic skeletons.

Using the **SkelCL** data-parallel skeleton library.

Targeting heterogeneous parallelism with **OpenCL**.

# The problem



Program performance is **not portable** across architectures.

Heterogeneous parallelism targets **numerous architectures**.

Algorithmic skeletons target **numerous problems**.

To achieve portable performance, we **need** autotuning.

# Progress so far

- ✓ Identify a set of one or more **tunable parameters** in SkelCL.
- ✓ Modify SkelCL so that these tunable parameters can be set at **runtime**.
- ✓ Create a set of representative skeleton program **benchmarks**.
- ✓ **Enumerate** the optimisation space of tunable parameters and benchmarks.
- **Search** the optimisation space for optimal parameter values.
- Use **machine learning** to predict optimal parameter values for unseen programs.

# SkelCL: Overview



OpenCL implementations of data parallel skeletons:

Map, Reduce, Scan, Zip, Stencil, AllPairs.

Each skeleton is a C++ class template, parameterised with OpenCL muscle functions.

Vector and Matrix containers implement lazy copying between host and device memories.

# SkelCL: how it works



```
; Initialise SkelCL.
skelcl_init(dev_type)

; Instantiate skeletons with user kernels.
Zip mult("int f(int x, int y) {return x*y}")
Reduce sum("int f(int x, int y) {return x+y}", 0)

; Call skeletons.
Vector result = sum(mult(vector_a, vector_b))

; Read result.
print result.first()
```

# SkelCL: how it works

```
; Initialise SkelCL.
skelcl_init(dev_type)

; Instantiate skeletons with user kernels.
Zip mult("int f(int x, int y) {return x*y}")
Reduce sum("int f(int x, int y) {return x+y}", 0)

; Call skeletons.
Vector result = sum(mult(vector_a, vector_b))

; Read result.
print result.first()
```

Initialise OpenCL devices

Prepare & compile device programs

Allocate space for data on hosts
Copy data to hosts

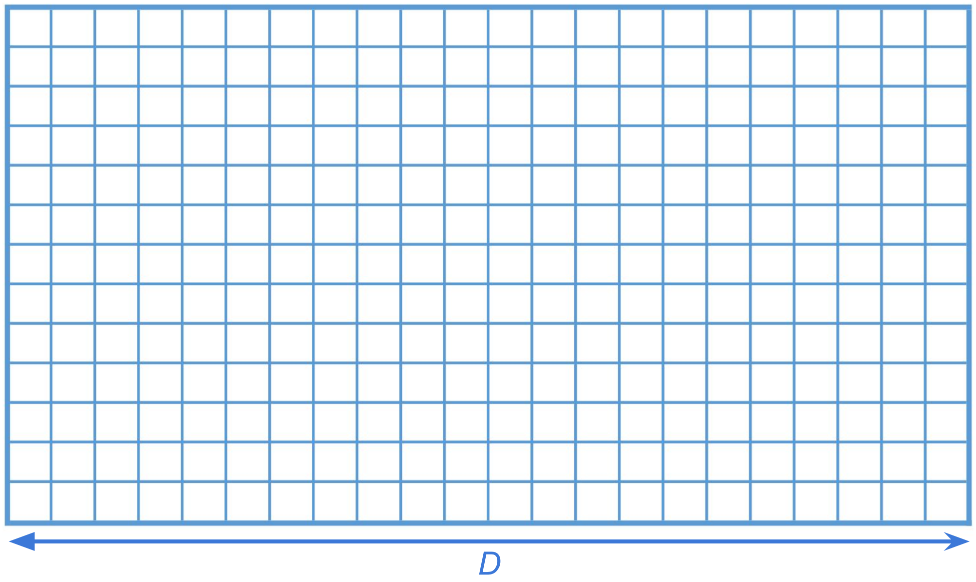
Enqueue jobs with devices

Wait until jobs have completed
Copy data from devices to host

# Stencil codes



Stencils operate on grids of **D**ata.

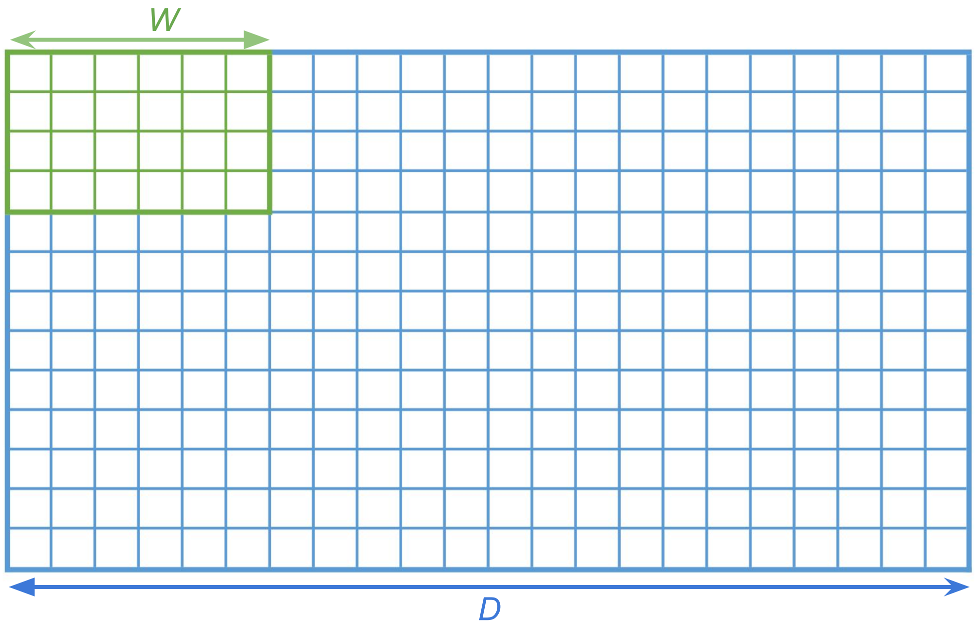


# Stencil codes



Stencils operate on grids of **D**ata.

Grids are decomposed into **Work groups**.



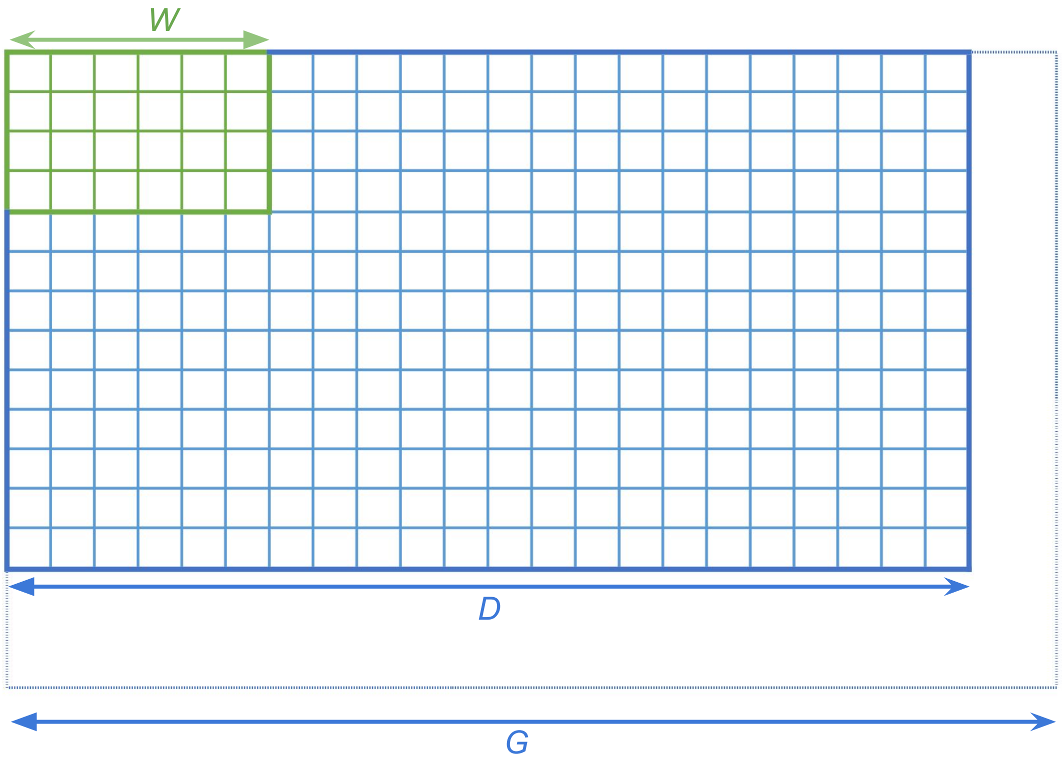
# Stencil codes



Stencils operate on grids of **D**ata.

Grids are decomposed into **Work groups**.

The size of the grid and work groups determine the **Global size**.



# Stencil codes

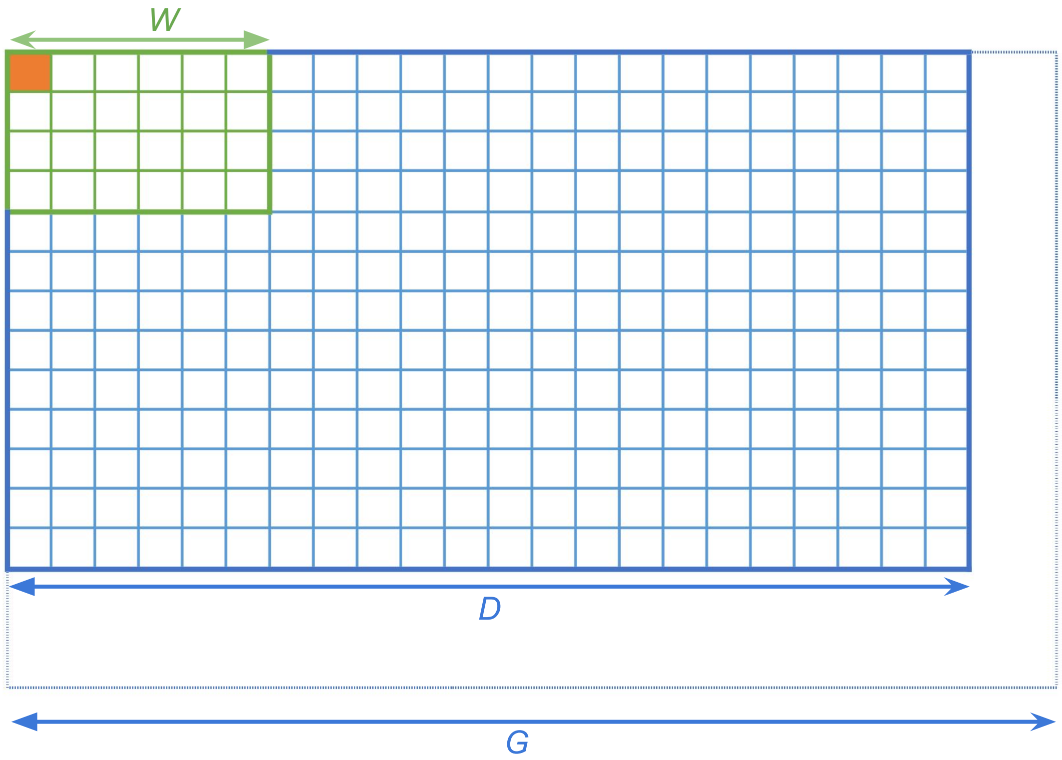


Stencils operate on grids of **D**ata.

Grids are decomposed into **Work groups**.

The size of the grid and work groups determine the **Global size**.

A **work item** operates on a single data element.



# Stencil codes



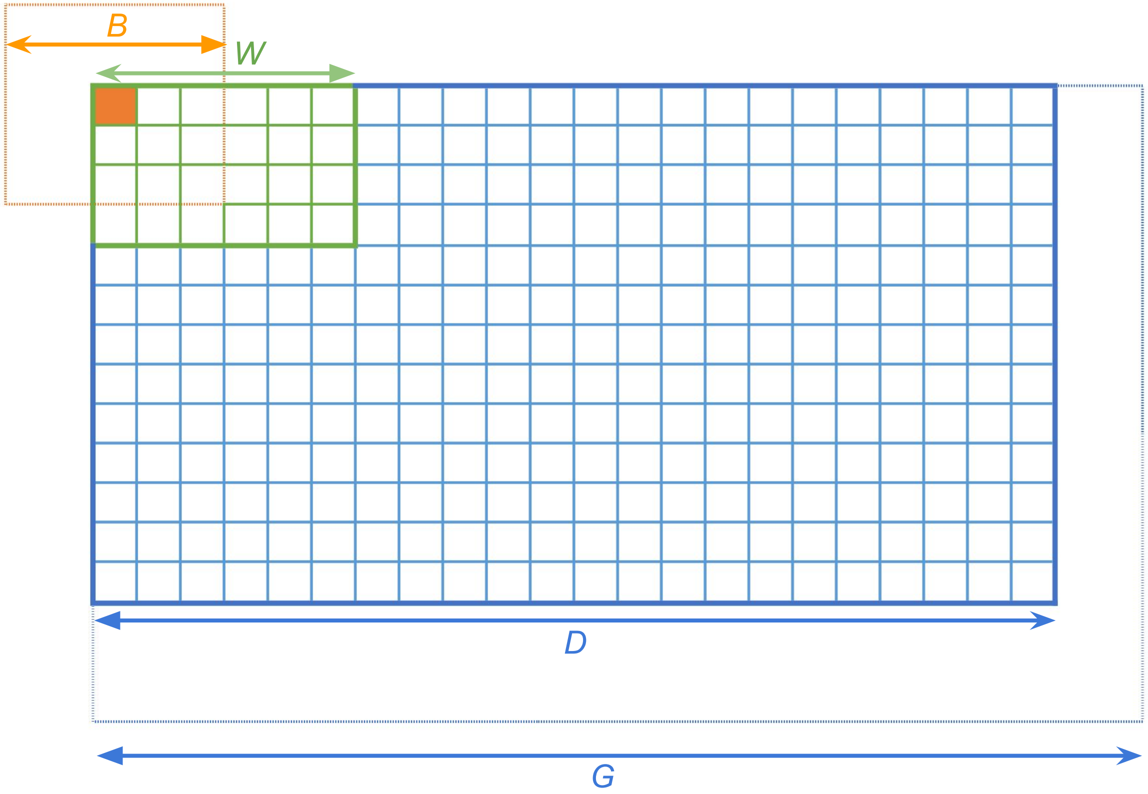
Stencils operate on grids of **D**ata.

Grids are decomposed into **Work groups**.

The size of the grid and work groups determine the **Global size**.

A **work item** operates on a single data element.

The **B**order region is the number of neighbouring elements accessed by a **work item**.



# Stencil codes



Stencils operate on grids of **D**ata.

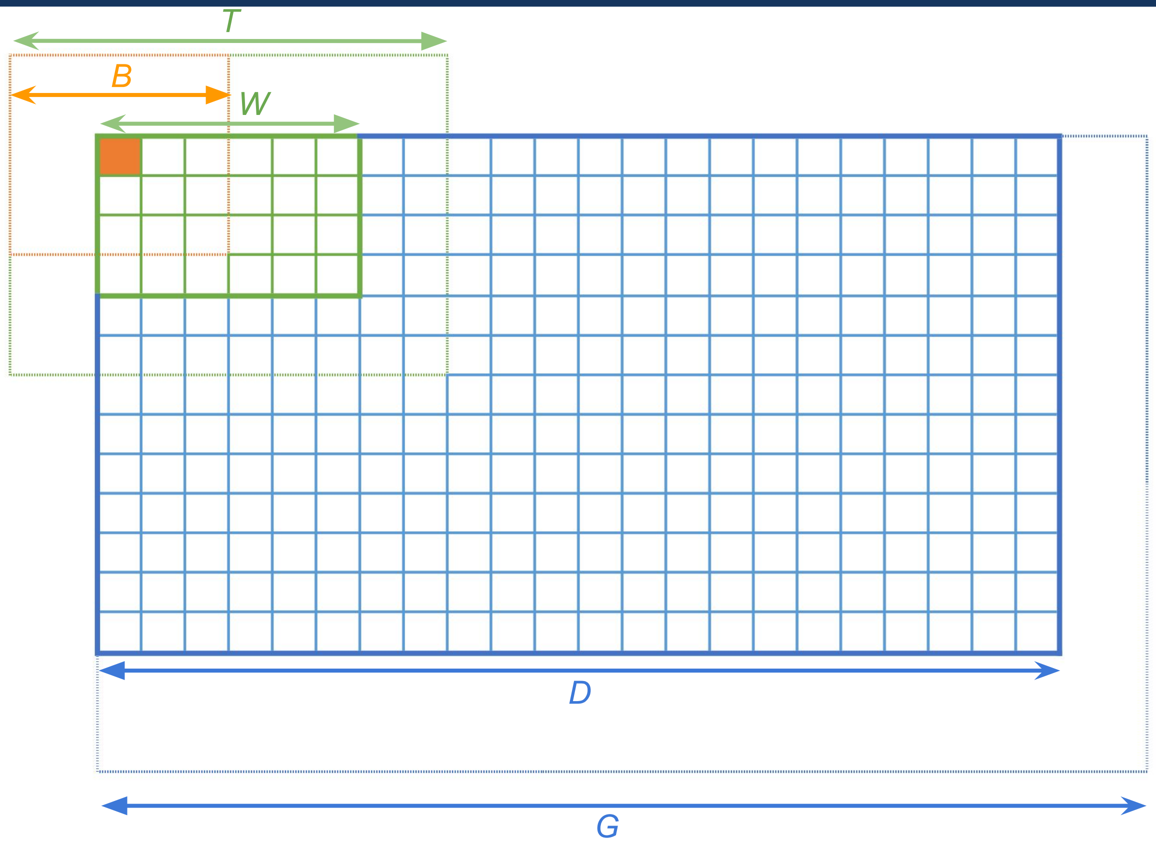
Grids are decomposed into **Work groups**.

The size of the grid and work groups determine the **Global size**.

A **work item** operates on a single data element.

The **B**order region is the number of neighbouring elements accessed by a **work item**.

The size of the work group and border region determines the **Tile size**.



# Autotuning stencil codes

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

The aim of autotuning is to generate a function which maps a set of feature vectors to optimal parameter values.

For stencil operations:

$$f : \boldsymbol{a}, \boldsymbol{k}, \boldsymbol{h}, \boldsymbol{d} \rightarrow \boldsymbol{p}$$

To explore this space, compare the performance of different ***p*** values for combinations of ***a***, ***k***, ***h***, ***d***.

| | |
|---|---|
| ***a*** | Architecture features |
| ***k*** | Kernel features |
| ***h*** | Border sizes (NESW) |
| ***d*** | Input data features |
| ***p*** | Parameter values |

# Autotuning stencil codes



Example ***p***arameters: work group size.

Changing work group size affects:

- Local memory allocation per workgroup.
- The number of  simultaneously active threads.
- Vectorisation for CPUs.

**BUT** values cannot be set arbitrarily:

- Hardware constraints limit maximum size.

| | |
|---|---|
| ***a*** | Architecture features |
| ***k*** | Kernel features |
| ***h*** | Border sizes (NESW) |
| ***d*** | Input data features |
| ***p*** | Parameter values |

# Exploring optimisation space



Defining an optimisation space for Stencil operations:

| | **Type** | **Variables** | **Values** |
|---|---|---|---|
| ***a*** | Architecture features | Host, execution device(s) | 10 device combinations |
| ***k*** | Kernel features | Simple or complex | 2 kernels |
| ***h*** | Border sizes (NESW) | North, East, South, West | 7 border sizes |
| ***d*** | Input data features | Grid size | 2 data sizes |
| ***p*** | Parameter values | Work group size | 9 combinations |

2520 unique configurations.

With hardware constraints: 1792 unique configurations.

# Exploring optimisation space



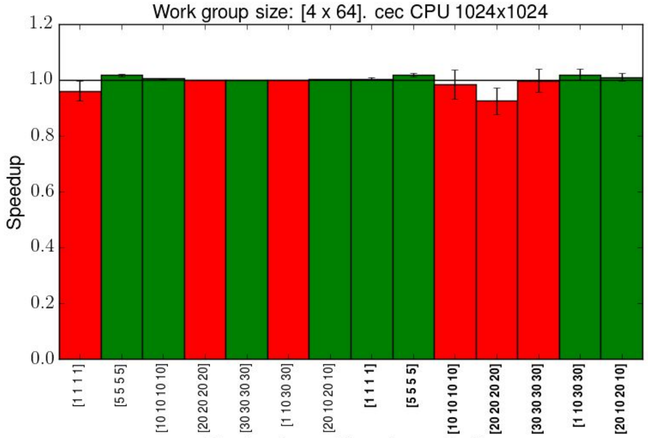
Using scripting to collect sound and reproducible results.

Experiment descriptor file: lists architectures, benchmarks, arguments, parameter values.

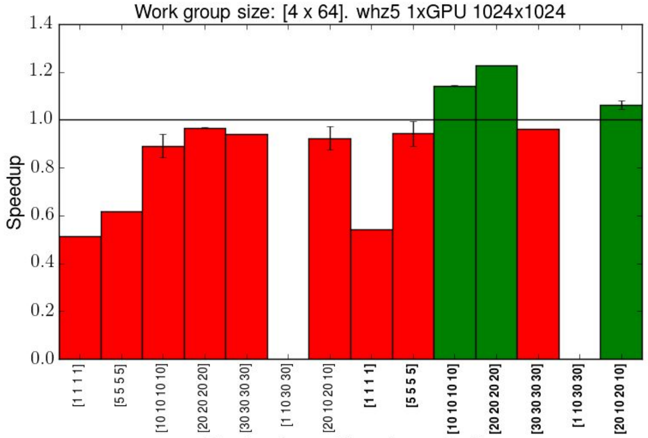
Script enumerates test cases, using fixed or dynamic sampling plans for statistical soundness.

```
{
  hosts: ["foo", "bar" ...],
  benchmarks: {
    GaussianBlur: {args: {...}},
    Mandelbrot: {args: {...}},
  },
  params: {
    LocalSizeR: [4, 8, 16],
    LocalSizeC: [32, 64]
  }
}
```

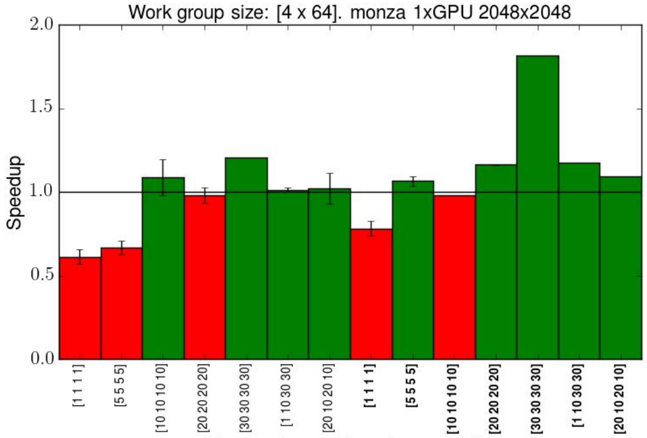
From the 1792 test cases, 252 are for this device, and 14 require samples...



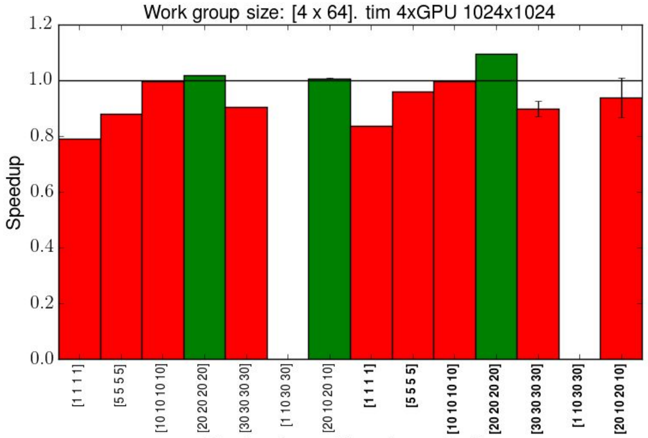

Average 10.0 iterations.



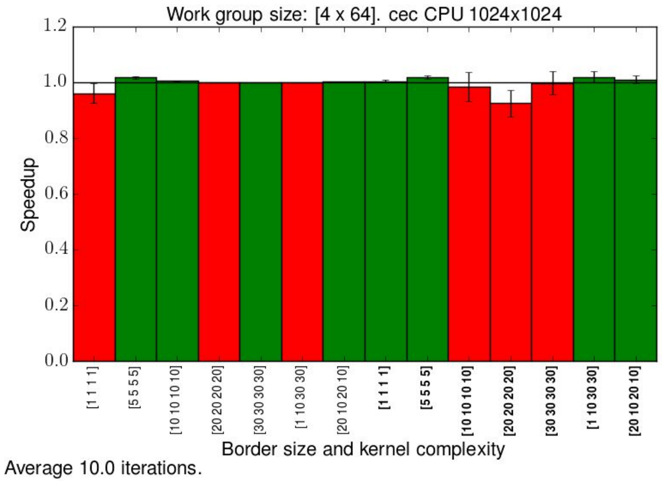

Average 8.0 iterations.

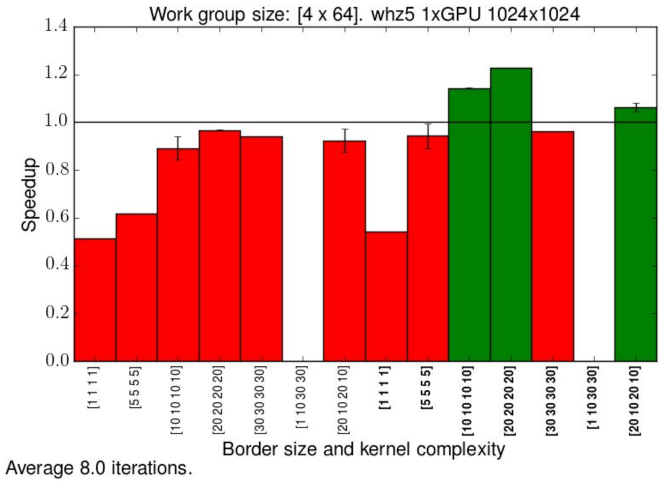



Average 10.0 iterations.

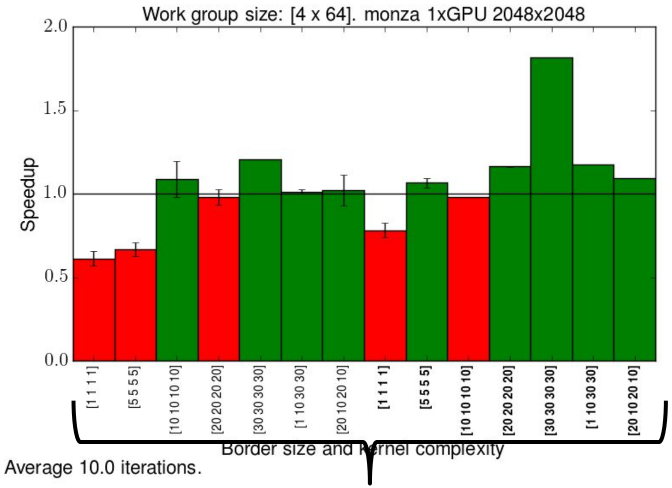


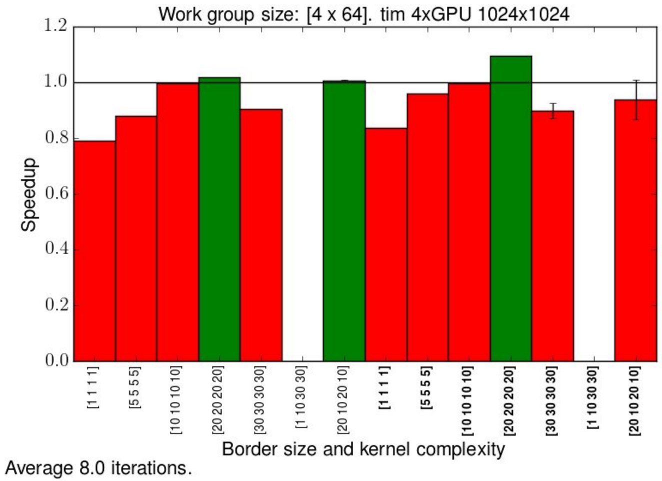

Average 8.0 iterations.

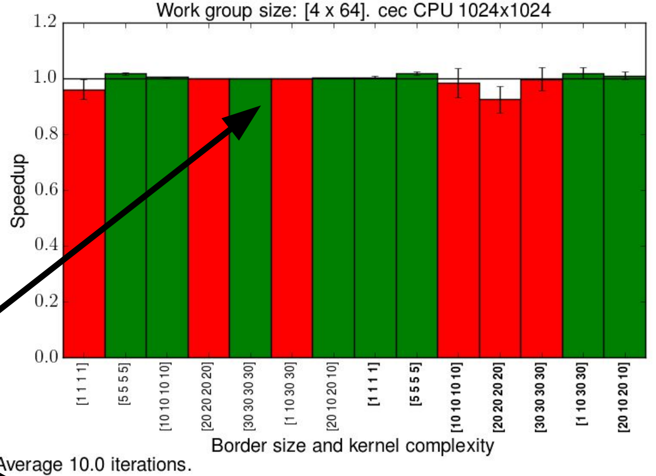


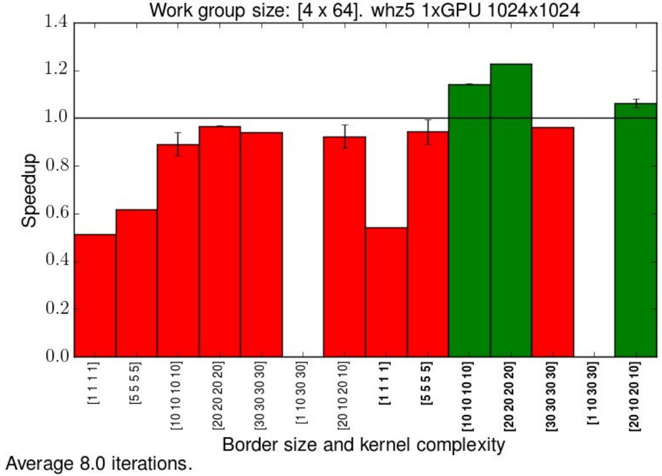




Performance across programs is non-uniform.

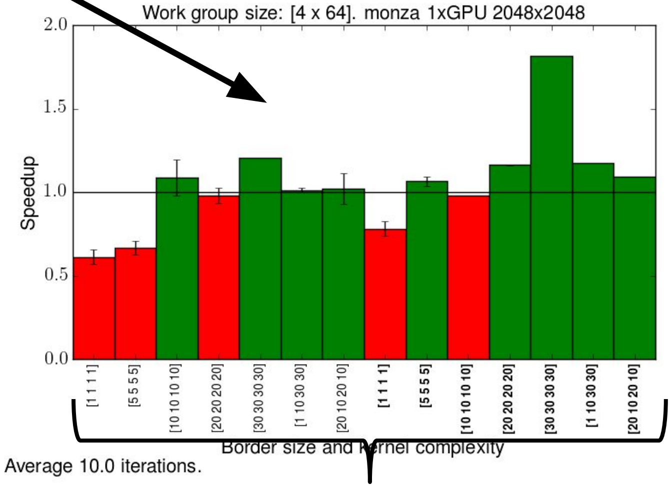


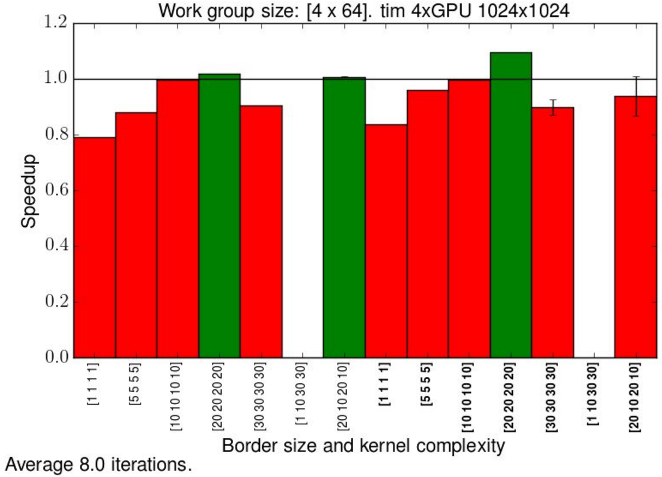






Performance across devices is non-uniform.

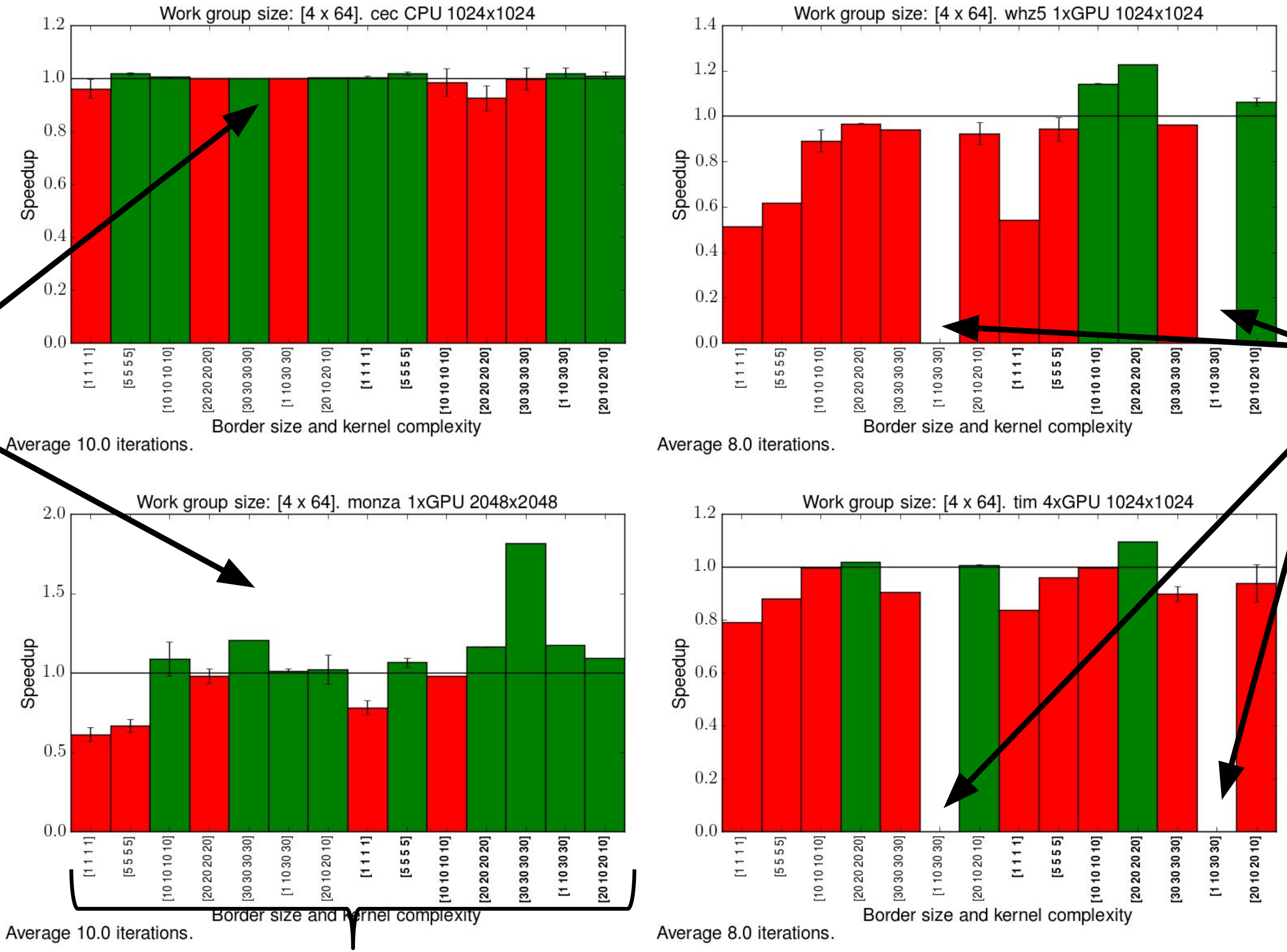


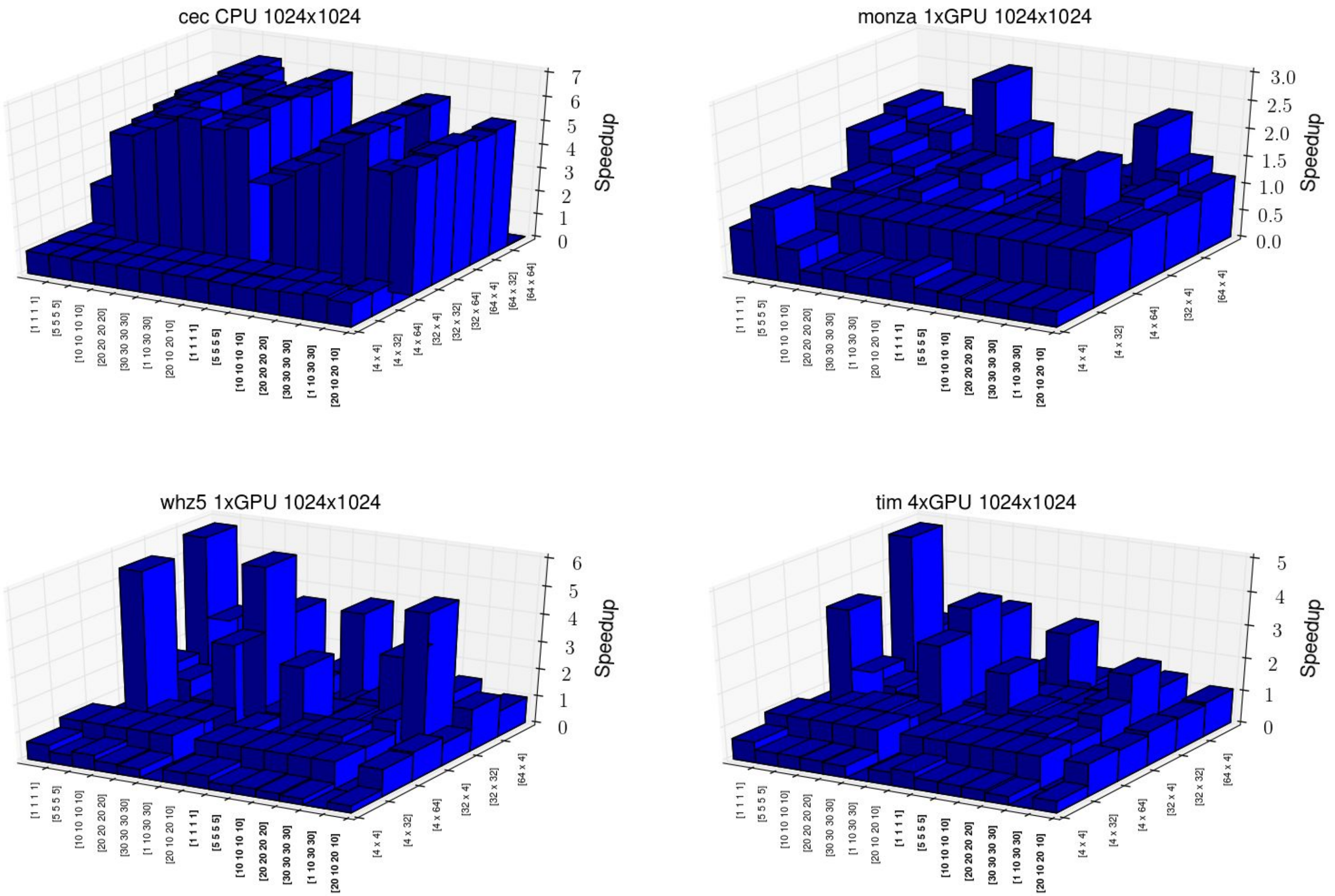

Performance across programs is non-uniform.

Performance across devices is non-uniform.

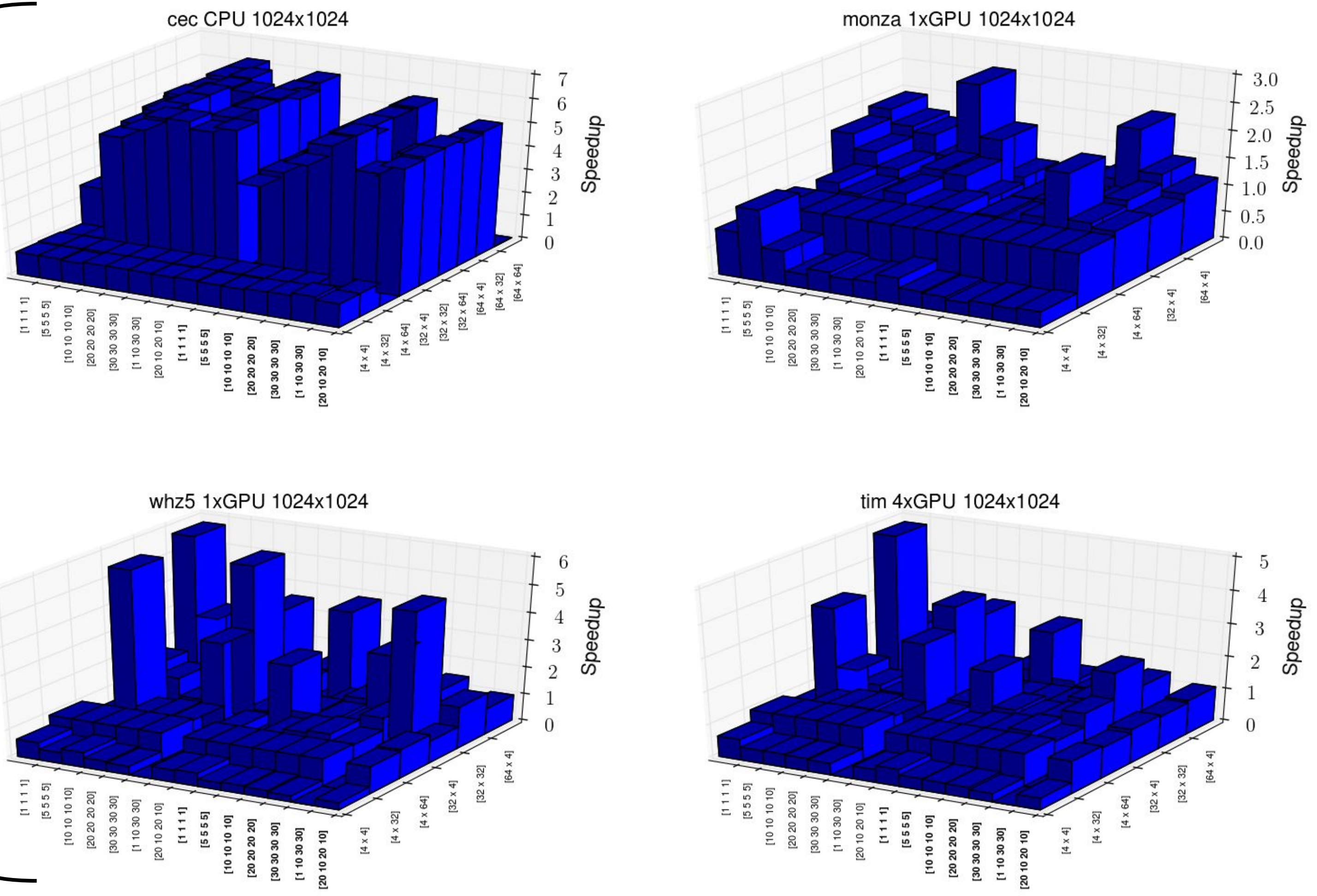
Space is non-continuous.

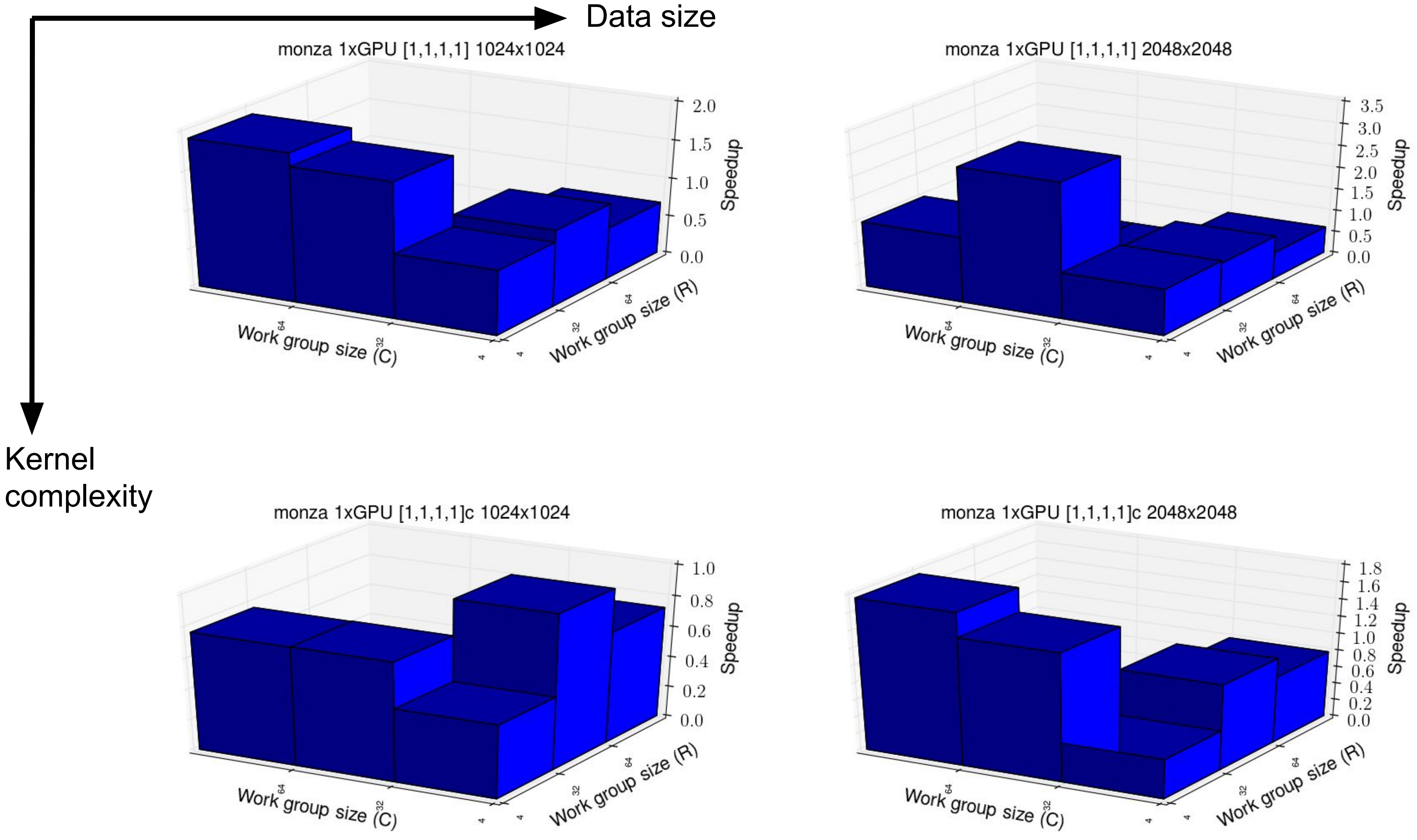



Performance across programs is non-uniform.


cec CPU 1024x1024
Speedup
0 1 2 3 4 5 6 7
[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]
**[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]**
[4 x 4] [4 x 32] [4 x 64] [32 x 4] [32 x 32] [32 x 64] [64 x 4] [64 x 32] [64 x 64]
monza 1xGPU 1024x1024
Speedup
0.0 0.5 1.0 1.5 2.0 2.5 3.0
[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]
**[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]**
[4 x 4] [4 x 32] [4 x 64] [32 x 4] [64 x 4]
whz5 1xGPU 1024x1024
Speedup
0 1 2 3 4 5 6
[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]
**[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]**
[4 x 4] [4 x 32] [4 x 64] [32 x 4] [32 x 32] [64 x 4]
tim 4xGPU 1024x1024
Speedup
0 1 2 3 4 5
[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]
**[1 1 1 1] [5 5 5 5] [10 10 10 10] [20 20 20 20] [30 30 30 30] [1 10 30 30] [20 10 20 10]**
[4 x 4] [4 x 32] [4 x 64] [32 x 4] [32 x 32] [64 x 4]

Performance across parameter values is non-uniform.





Data size
Kernel complexity
monza 1xGPU [1,1,1,1] 1024x1024
monza 1xGPU [1,1,1,1] 2048x2048
monza 1xGPU [1,1,1,1]c 1024x1024
monza 1xGPU [1,1,1,1]c 2048x2048
Speedup
Work group size (C)
Work group size (R)
64
32
4
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0.2
0.4
0.6
0.8
1.2
1.4
1.6
1.8

# Autotuning stencil codes



**172** oracle configurations. Default parameter values optimal for **2.29%** of cases.

Average oracle speedup **2.393x** (max **7.192x**).

Cannot statically tune: hardware constraints invalidate most optimal value for some cases.

Use **machine learning** to predict best values for new configurations.

Training data created from results of optimisation space exploration:

- **Features**: vectors of 11 explanatory variables.
- **Labels**: oracle workgroup size values.

# Autotuning stencil codes



**Decision tree** classifier:

- Simple to reason about and implement.
- Use Weka's J48 algorithm (default params).
- Script parses output and implements tree.

Results from 10 fold cross validation:

- Optimal parameter values **76.57%** of time.
- Average speedup of **2.309x**.
- Min **0.699x**, max **7.192x**.
- Performance is **95.83%** of oracle.

```
classify(x):
  if x.DevType == GPU:
    if x.BorderEast <= 10: return (64, 4)
    if x.BorderEast > 10:
      if x.Hostname == cec: return (32, 32)
      if x.Hostname == monza: return (64, 4)
      if x.Hostname == florence: return (32, 32)
      if x.Hostname == whz5: return (32, 32)
      if x.Hostname == tim:
        if x.Complexity == 0:
          if x.BorderNorth <= 20: return (64, 4)
          if x.BorderNorth > 20: return (32, 32)
        if x.Complexity == 1: return (32, 32)
  if x.DevType == CPU:
    if x.BorderNorth <= 20:
      if x.BorderSouth <= 10:
      ...
```

# Moving forward



Something I should improve upon: tackling “simple” cases first, then scaling up.

My approach was to complete one big block of work each for:

Developing benchmarks.

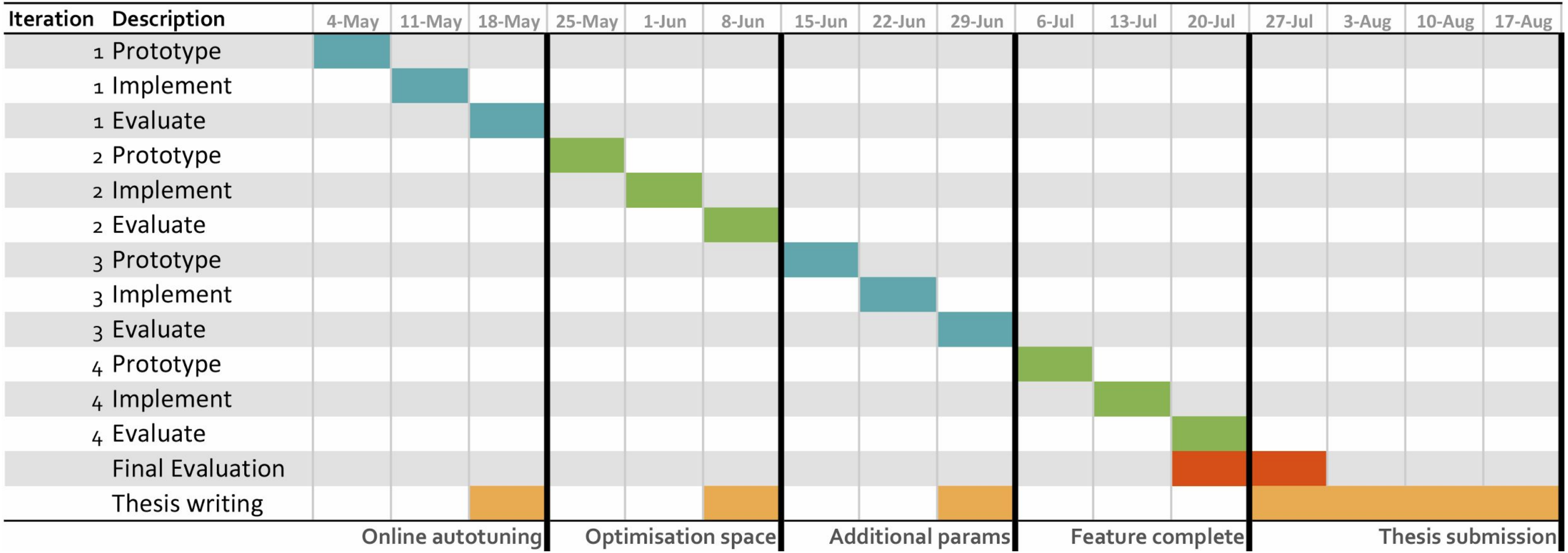
Creating tunable parameters.

Enumerating optimisation space.

Developing an autotuner.

For the remainder of the project (& PhD phase), use a workflow with small, incremental iterations.

# Moving forward



# Moving forward

**Iteration 1** - Online autotuning

- Runtime search
- Machine learning guided search
- Online machine learning

**Iteration 2** - Optimisation space

- Multi-label classification
- Auto-generated kernels
- Kernel feature extraction

**Iteration 3** - Additional params

- Stencil iterations between swaps
- MapOverlap vs Stencil
- Execution device and count

**Iteration 4** - Feature complete

- Integration & testing
- Evaluation

# Conclusions



There are a number of parameters which affect the performance of SkelCL. Performance depends on: muscle functions, input data, number of iterations, and architecture.

Experimental results for parameters. For a simple case, offline machine learning can achieve 96% of oracle performance for 2.3x speedup.

Remaining time spent on 4 short iterations targeting: online autotuning, increasing the number of parameters, and automatically generating benchmark programs.

EPSRC Centre for Doctoral Training in Pervasive Parallelism

# End of presentation

# SkelCL: how it works



```
; Initialise SkelCL.
skelcl_init(dev_type)

; Instantiate skeletons with user kernels.
Zip mult(“int f(int x, int y) {return x*y}”)
Reduce sum(“int f(int x, int y) {return x+y}”, 0)

; Call skeletons.
Vector result = sum(mult(vector_a, vector_b))

; Read result.
print result.first()
```

Initialise OpenCL devices

Prepare & compile
device programs

Allocate space for data on hosts
Copy data to hosts

Enqueue jobs with devices

Wait until jobs have completed
Copy data from devices to host

# Profiling SkelCL applications



Time types:

| | |
|---|---|
| ***c*** | Kernel compilation times |
| ***p*** | Skeleton prepare times |
| ***u*** | Host -> Device |
| ***k*** | Device execution times |
| ***d*** | Device -> Host |
| ***s*** | Devices <-> Host (sync) |

*Obtained from OpenCL events.*

Prepare & compile
device programs

Allocate space for data on hosts
Copy data to hosts

Enqueue jobs with devices

Wait until jobs have completed
Copy data from devices to host

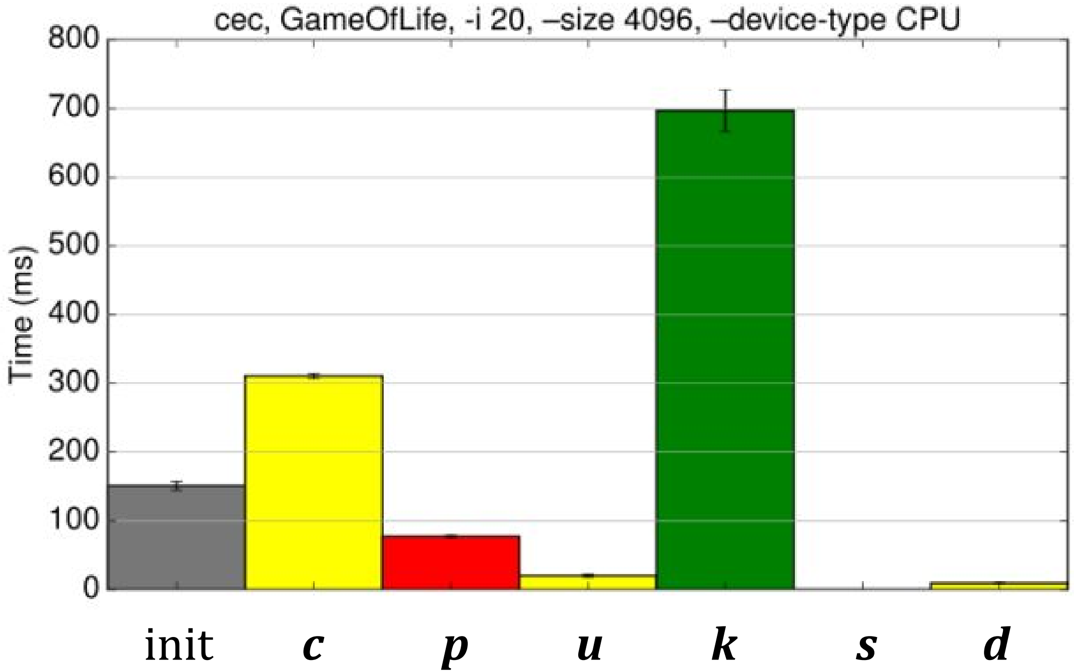
# Profiling SkelCL applications



Time types:

| | |
|---|---|
| ***c*** | Kernel compilation times |
| ***p*** | Skeleton prepare times |
| ***u*** | Host -> Device |
| ***k*** | Device execution times |
| ***d*** | Device -> Host |
| ***s*** | Devices <-> Host (sync) |

*Obtained from OpenCL events.*




Times: no-init: **1115.25** ms, no-build: **804.25** ms. device: **726.55** ms.
ID: `c9b9bcf6c928d805a0730f1789fe205b2f39fc09`. 10 samples.

# Profiling SkelCL applications



Time types:

| | |
|---|---|
| ***c*** | Kernel compilation times |
| ***p*** | Skeleton prepare times |
| ***u*** | Host -> Device |
| ***k*** | Device execution times |
| ***d*** | Device -> Host |
| ***s*** | Devices <-> Host (sync) |

*Obtained from OpenCL events.*

Approximating end-to-end time (~95% accurate):

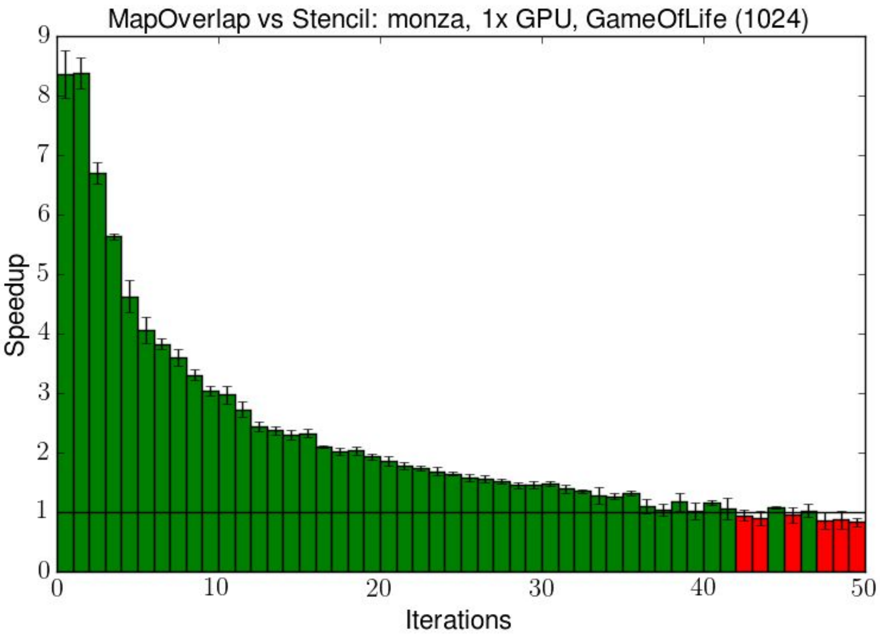
$$t \approx \underbrace{\sum_{i=1}^{n} \mathbf{1}\boldsymbol{c}_i + \mathbf{1}\boldsymbol{p} + \mathbf{1}\boldsymbol{s}}_{\text{Host}} + \underbrace{\frac{\sum_{i=1}^{n} \mathbf{1}\boldsymbol{u}_i + \mathbf{1}\boldsymbol{k}_i + \mathbf{1}\boldsymbol{d}_i}{n}}_{\text{Devices}}$$

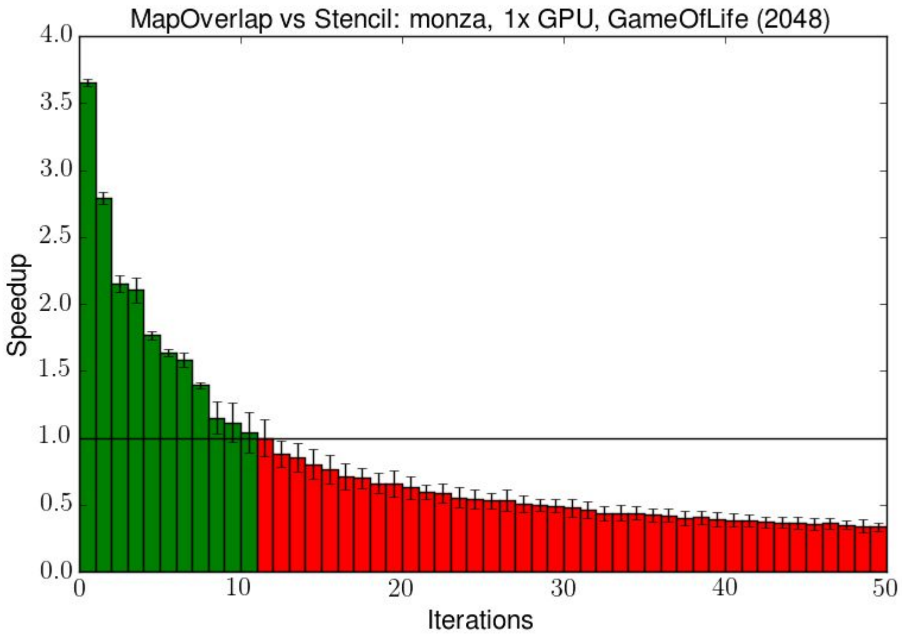
Measuring end-to-end time:

```
start timer
; Returns instantly:
result = skeleton(input)
; Force copy:
copy result to host
stop timer
```

# Tuning iterative stencil

Trading device memory bandwidth against kernel complexity.

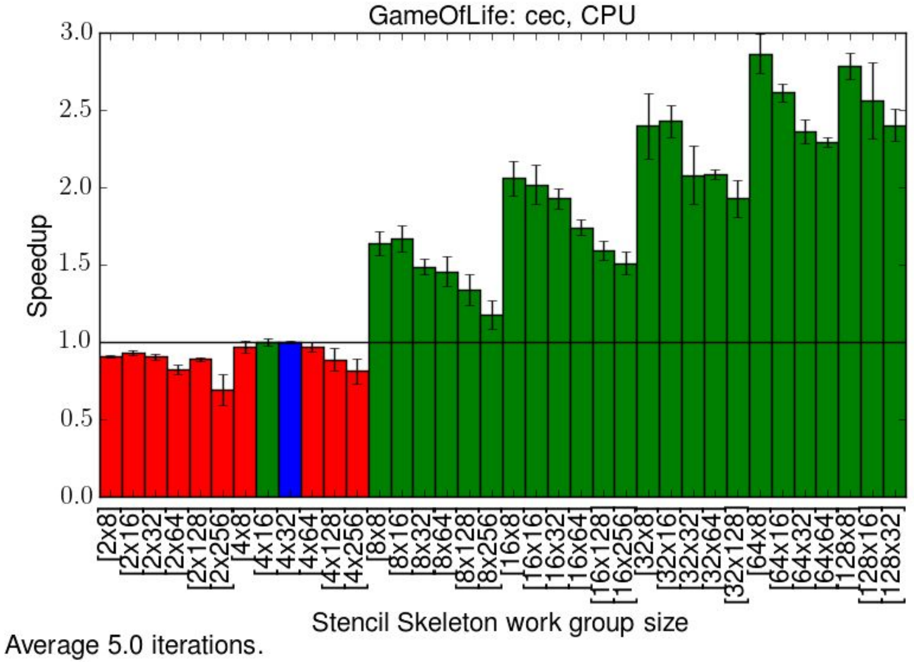


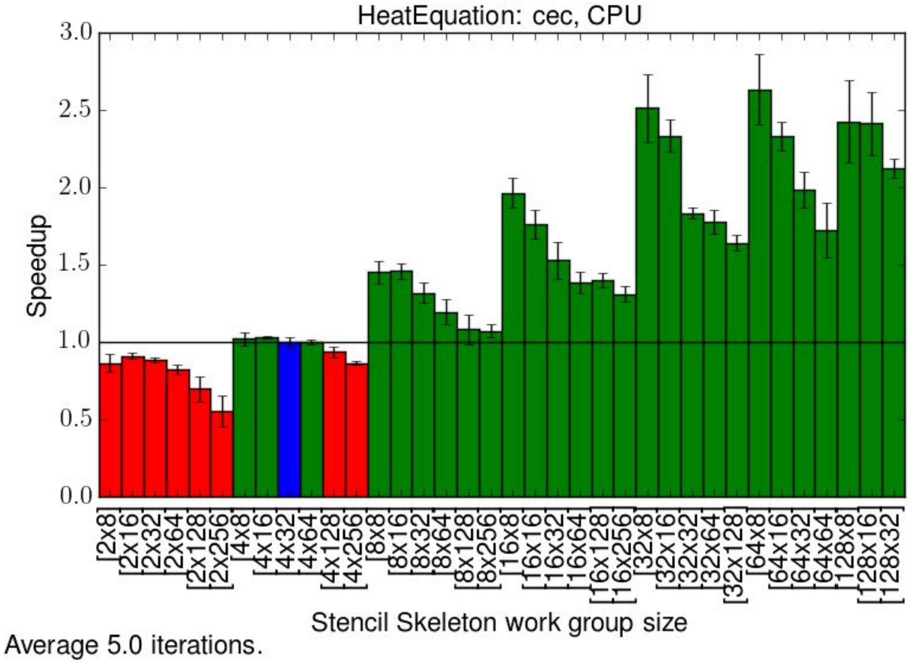



# The need for generated kernels

EPSRC Centre for Doctoral Training in Pervasive Parallelism

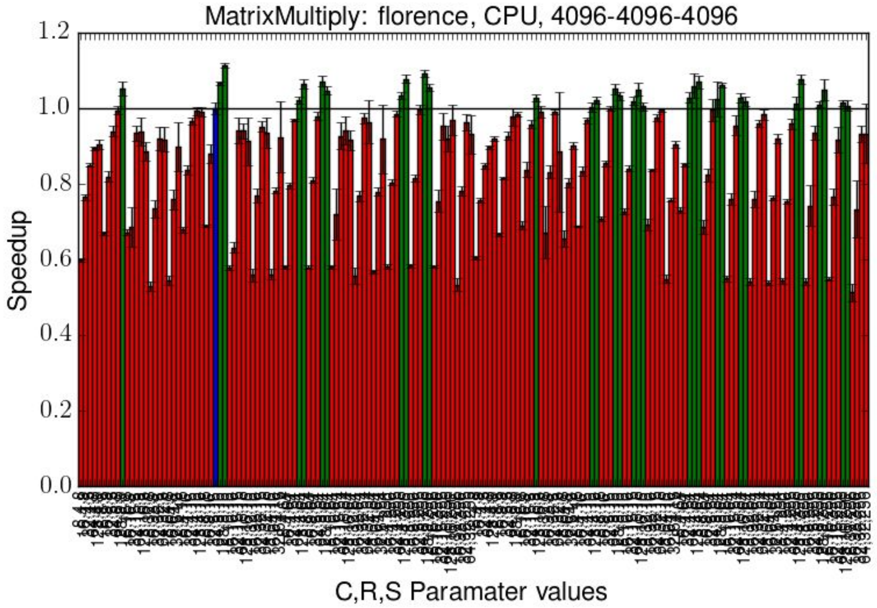
Benchmarks are too similar, no inter-program tuning necessary:



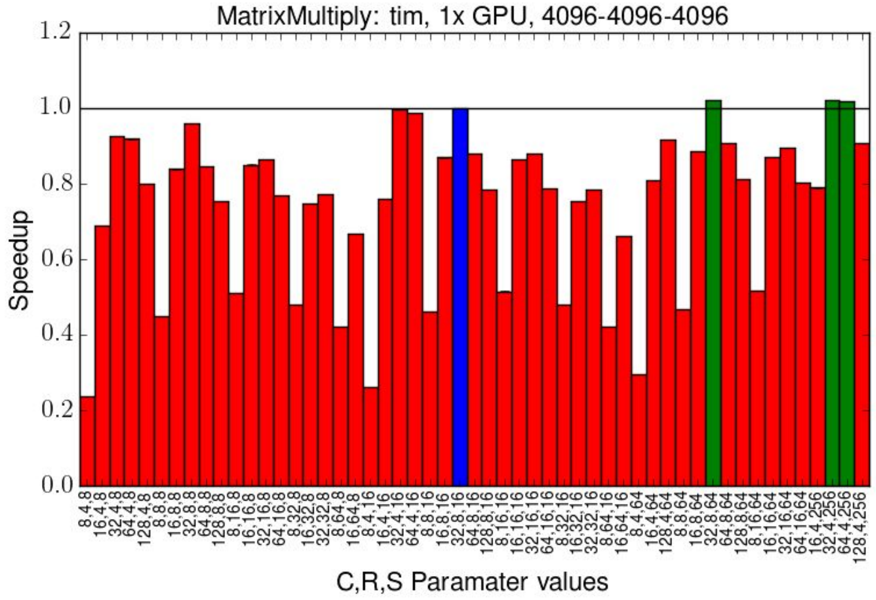



# Tuning AllPairs

Too little room for improvement, no tuning necessary:




Average 5.0 iterations.




Average 4.0 iterations.