# Analyzing and improving performance portability of OpenCL applications via auto-tuning

**James Price & Simon McIntosh-Smith**
University of Bristol - High Performance Computing Group
http://uob-hpc.github.io

**Funded in part by Imagination Technologies**

# Overview

- Performance portability is one of the key concerns for developers targeting many different architectures
- Current work in this area has provided mixed results
- Here we propose a method of rigorously analyzing performance portability by exploiting the black-box nature of auto-tuning
- We also present a simple technique that can improve performance portability when using auto-tuning

# Analysis approach

- Expose set of possible implementation decisions as tuning options
- Dynamically generate kernels to provide greater flexibility
- Run auto-tuner to optimize the kernel for each device individually
- This produces a set of architecture-specific kernels
- We then run each architecture-specific kernel on every other device and measure efficiency

# Benchmark #1: Jacobi method

- Work-group size / parallel decomposition
- Memory layout / memory access pattern
- Loop unrolling
- *+ vs mad vs fma
- Branching vs masks
- Division by diagonal (inline or precompute)
- Address spaces of input vectors
- Embedding values into kernel as constants

Solve $A\boldsymbol{x} = \boldsymbol{b}$

↓

Split matrix A into diagonal D and remainder R

↓

$$\boldsymbol{x}_{i+1} = D^{-1}(\boldsymbol{b} - R\boldsymbol{x}_i)$$

# Benchmark #2: Bilateral filter

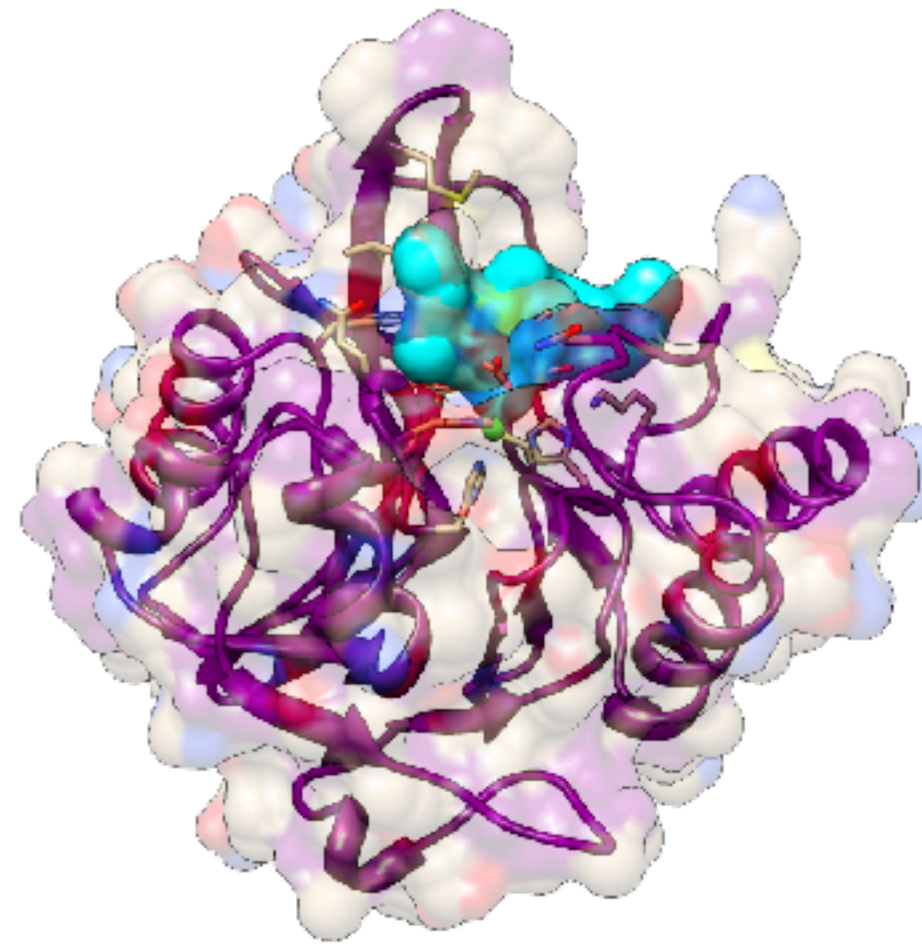
- Work-group size
- Tile size / tile layout
- Prefetch pixels into private/local memory
- Buffers vs images
- *+ vs mad vs fma
- Loop interchange
- Native math functions
- Embedding values into kernel as constants

$$O(x,y) = \frac{\sum_{i=-r}^{r} \sum_{j=-r}^{r} I(i,j) \cdot W(x,y,i,j)}{\sum_{i=-r}^{r} \sum_{j=-r}^{r} W(x,y,i,j)}$$

$$W(x,y,i,j) = e^{(-\frac{\sqrt{i^2+j^2}}{2\sigma_d} - \frac{||(I(x,y)-I(x+i,y+j)||}{2\sigma_r})}$$

# Benchmark #3: BUDE

- Parallel decomposition / work-group size
- Data layout / memory access patterns
- Loop interchange
- Address space of molecules / forcefield
- Precomputing forcefield coefficients
- Native math functions
- Embedding values into kernel as constants



# Devices

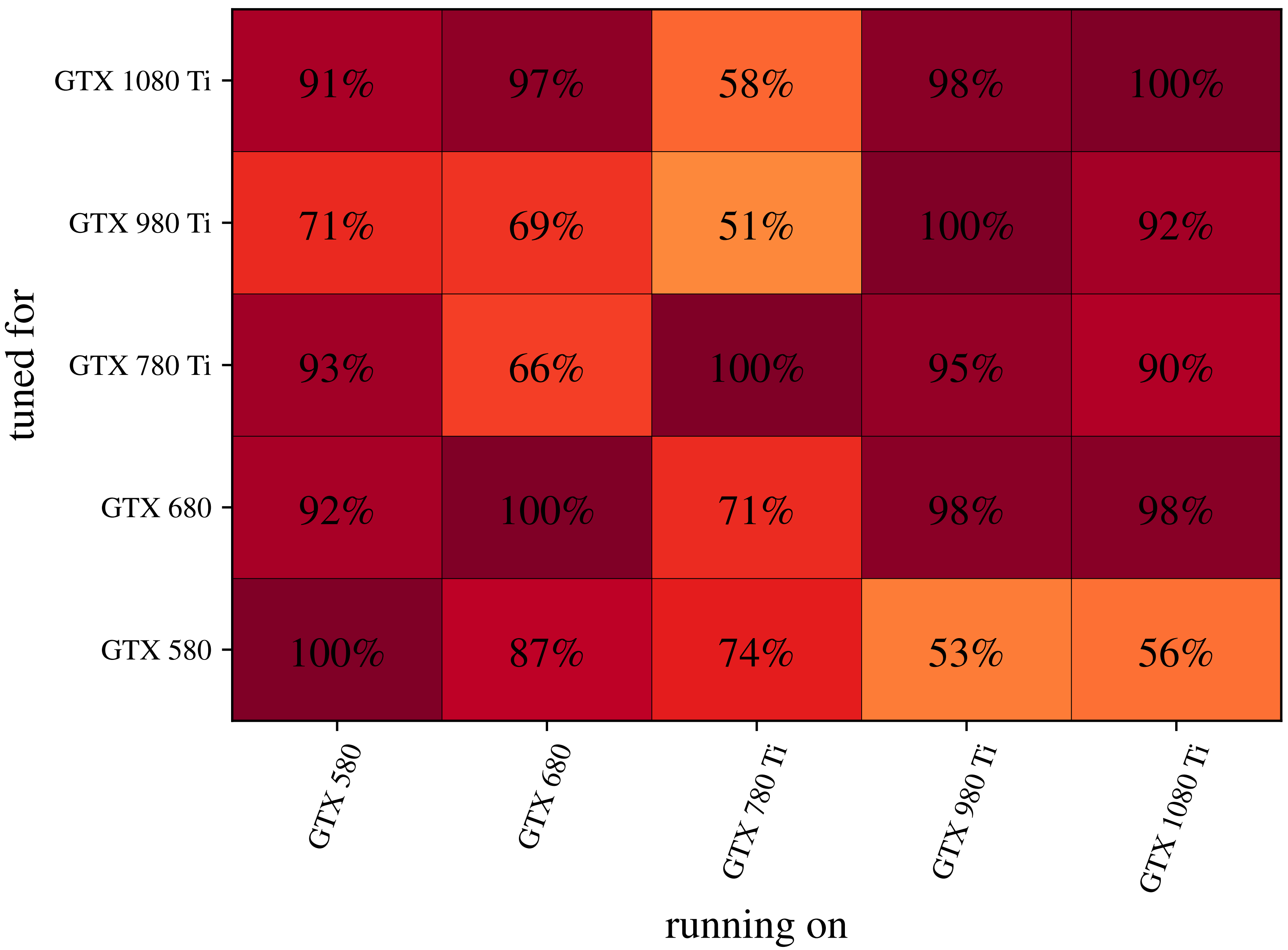
| Vendor | Product | Architecture | Compute units |
| --- | --- | --- | --- |
| NVIDIA | GeForce GTX 580 | Fermi (GF110) | 16 |
| | GeForce GTX 680 | Kepler (GK104) | 8 |
| | GeForce GTX 780 Ti | Kepler (GK110) | 15 |
| | GeForce GTX 980 Ti | Maxwell (GM200) | 22 |
| | GeForce GTX 1080 Ti | Pascal (GP102) | 28 |
| AMD | Radeon HD 7970 | Tahiti | 32 |
| | Radeon R9 290X | Hawaii | 44 |
| | Radeon R9 Furyx | Fiji | 64 |
| | Radeon RX 480 | Ellesmere | 36 |
| Intel | Core i5-3550 | Ivy Bridge | 4 |
| | Core i5-4590 | Haswell | 4 |
| | Core i5-6600 | Skylake | 4 |

# Jacobi efficiencies

| tuned for \ running on | GTX 580 | GTX 680 | GTX 780 Ti | GTX 980 Ti | GTX 1080 Ti | HD 7970 | R9 290X | R9 Fury X | RX 480 | Ivy Bridge CPU | Haswell CPU | Skylake CPU |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Skylake CPU | 90% | 69% | 93% | 90% | 90% | 45% | 28% | 14% | 21% | 2% | 99% | 100% |
| Haswell CPU | 12% | 16% | 16% | 39% | 43% | 37% | 20% | 11% | 17% | 2% | 100% | 97% |
| Ivy Bridge CPU | 21% | 14% | 18% | 41% | 40% | 19% | 14% | 6% | 13% | 100% | 53% | 44% |
| RX 480 | 87% | 74% | 78% | 95% | 93% | 99% | 99% | 97% | 100% | 2% | 93% | 81% |
| R9 Fury X | 24% | 20% | 20% | 23% | 37% | 100% | 99% | 100% | 99% | 62% | 80% | 65% |
| R9 290X | 91% | 79% | 84% | 97% | 97% | 99% | 100% | 93% | 100% | 2% | 70% | 57% |
| HD 7970 | 23% | 19% | 20% | 21% | 33% | 100% | 100% | 99% | 100% | 73% | 85% | 71% |
| GTX 1080 Ti | 91% | 97% | 58% | 98% | 100% | 68% | 38% | 28% | 62% | 1% | 5% | 4% |
| GTX 980 Ti | 71% | 69% | 51% | 100% | 92% | 53% | 33% | 49% | 36% | 5% | 5% | 5% |
| GTX 780 Ti | 93% | 66% | 100% | 95% | 90% | 68% | 33% | 32% | 62% | 5% | 5% | 4% |
| GTX 680 | 92% | 100% | 71% | 98% | 98% | 55% | 38% | 26% | 57% | 1% | 5% | 4% |
| GTX 580 | 100% | 87% | 74% | 53% | 56% | X | X | X | X | 1% | 5% | 4% |

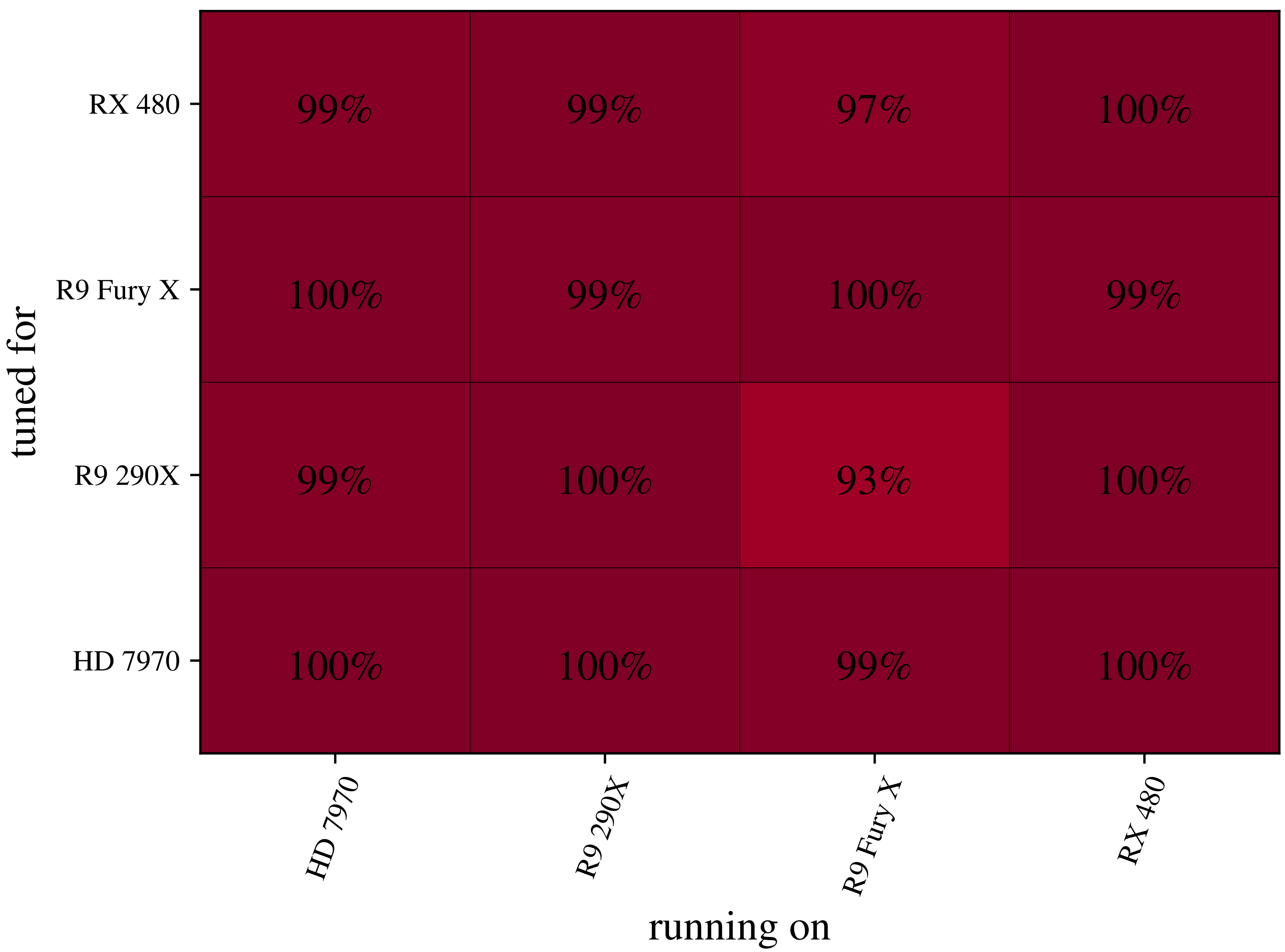
running on

# Jacobi - NVIDIA



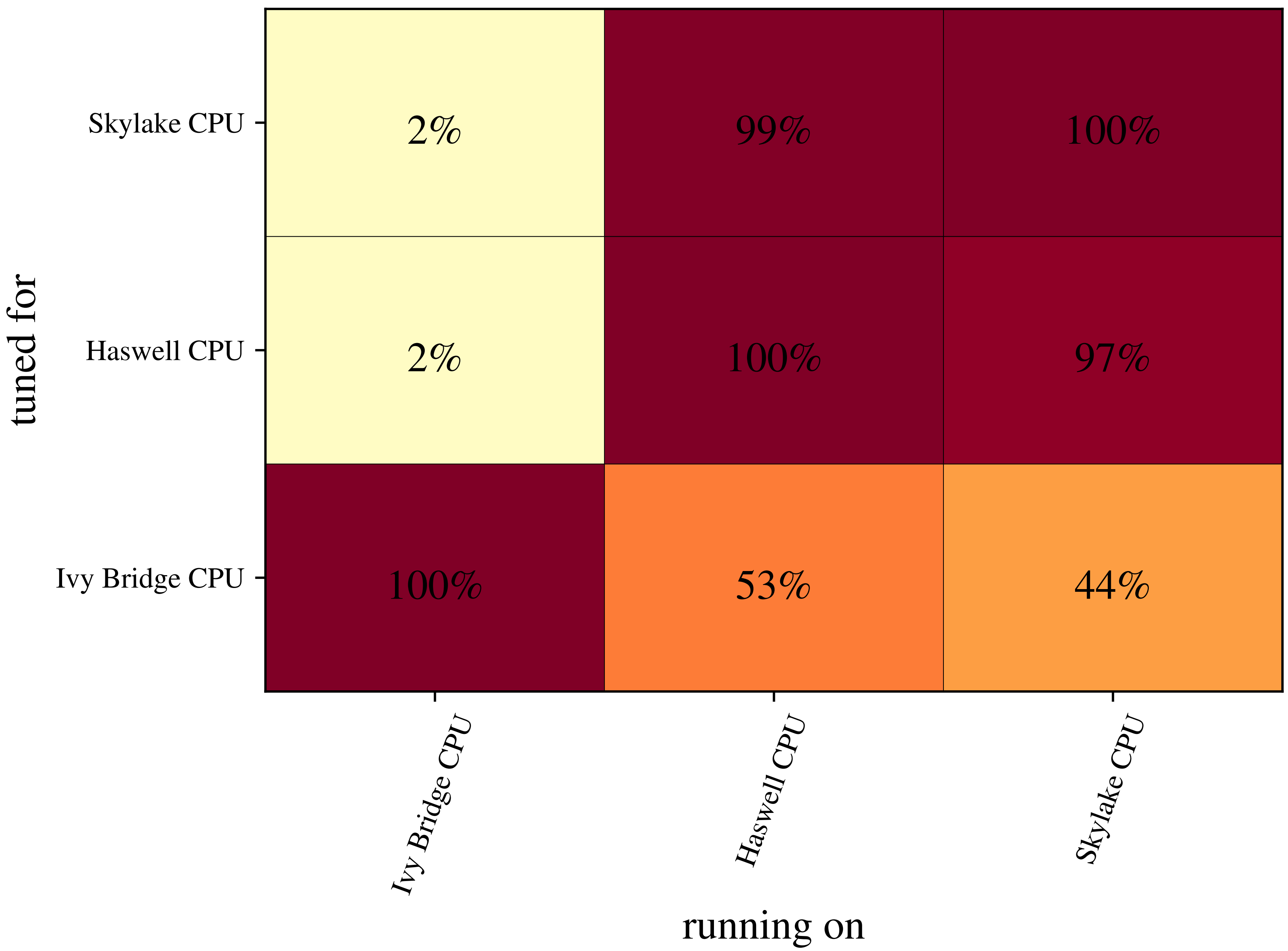

- Work-group size differs between devices
- Other drops due to address spaces and memory access pattern

# Jacobi - AMD



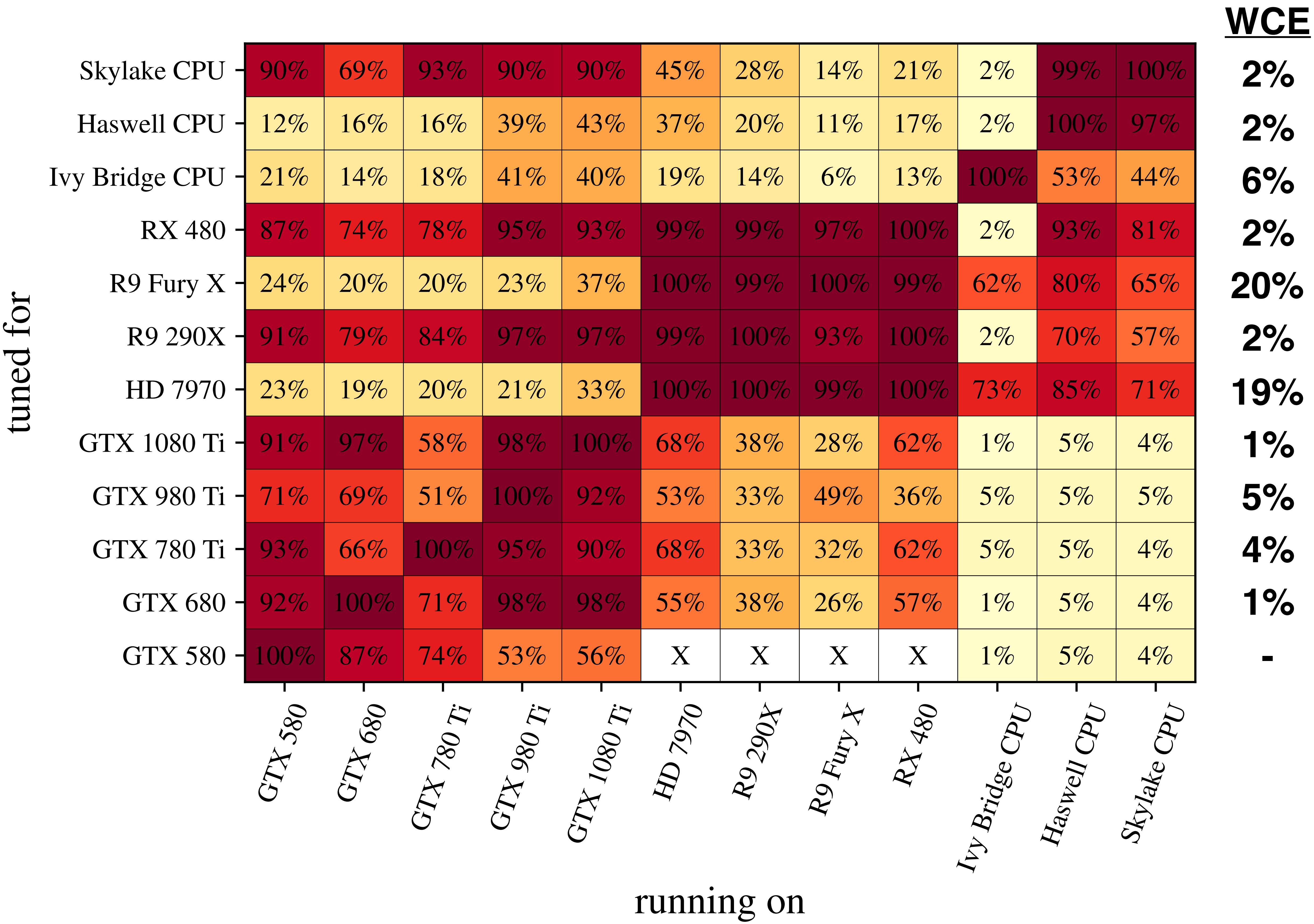

- Most parameters uniform between all AMD devices
- FuryX suffers a little with expensive division

# Jacobi - Intel




- Ivy Bridge performs poorly with **fma** builtin
- Vectorisation width differs

# Jacobi efficiencies



# Bilateral efficiencies

| tuned for \ running on | GTX 580 | GTX 680 | GTX 780 Ti | GTX 980 Ti | GTX 1080 Ti | R9 290X | R9 Fury X | RX 480 | Ivy Bridge CPU | Haswell CPU | Skylake CPU | **WCE** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Skylake CPU | 19% | 13% | 18% | 17% | 12% | 11% | 8% | 9% | 98% | 100% | 100% | **8%** |
| Haswell CPU | 19% | 13% | 18% | 17% | 12% | 11% | 8% | 9% | 98% | 100% | 100% | **8%** |
| Ivy Bridge CPU | 8% | 6% | 7% | 7% | 7% | 6% | 5% | 5% | 100% | 99% | 99% | **5%** |
| RX 480 | 70% | 70% | 69% | 84% | 87% | 100% | 100% | 100% | 29% | 27% | 26% | **26%** |
| R9 Fury X | 70% | 71% | 76% | 85% | 87% | 100% | 100% | 100% | 28% | 27% | 26% | **26%** |
| R9 290X | 70% | 68% | 73% | 85% | 86% | 100% | 100% | 100% | 28% | 27% | 25% | **25%** |
| GTX 1080 Ti | 93% | 90% | 97% | 97% | 100% | X | X | X | 14% | 72% | 73% | **-** |
| GTX 980 Ti | 100% | 97% | 99% | 100% | 98% | 35% | 31% | 37% | 14% | 72% | 71% | **14%** |
| GTX 780 Ti | 95% | 96% | 100% | 98% | 98% | 37% | 37% | 45% | 74% | 71% | 73% | **37%** |
| GTX 680 | 96% | 100% | 99% | 98% | 97% | 37% | 38% | 45% | 74% | 72% | 74% | **37%** |
| GTX 580 | 100% | 98% | 99% | 100% | 99% | 35% | 31% | 29% | 14% | 73% | 74% | **14%** |

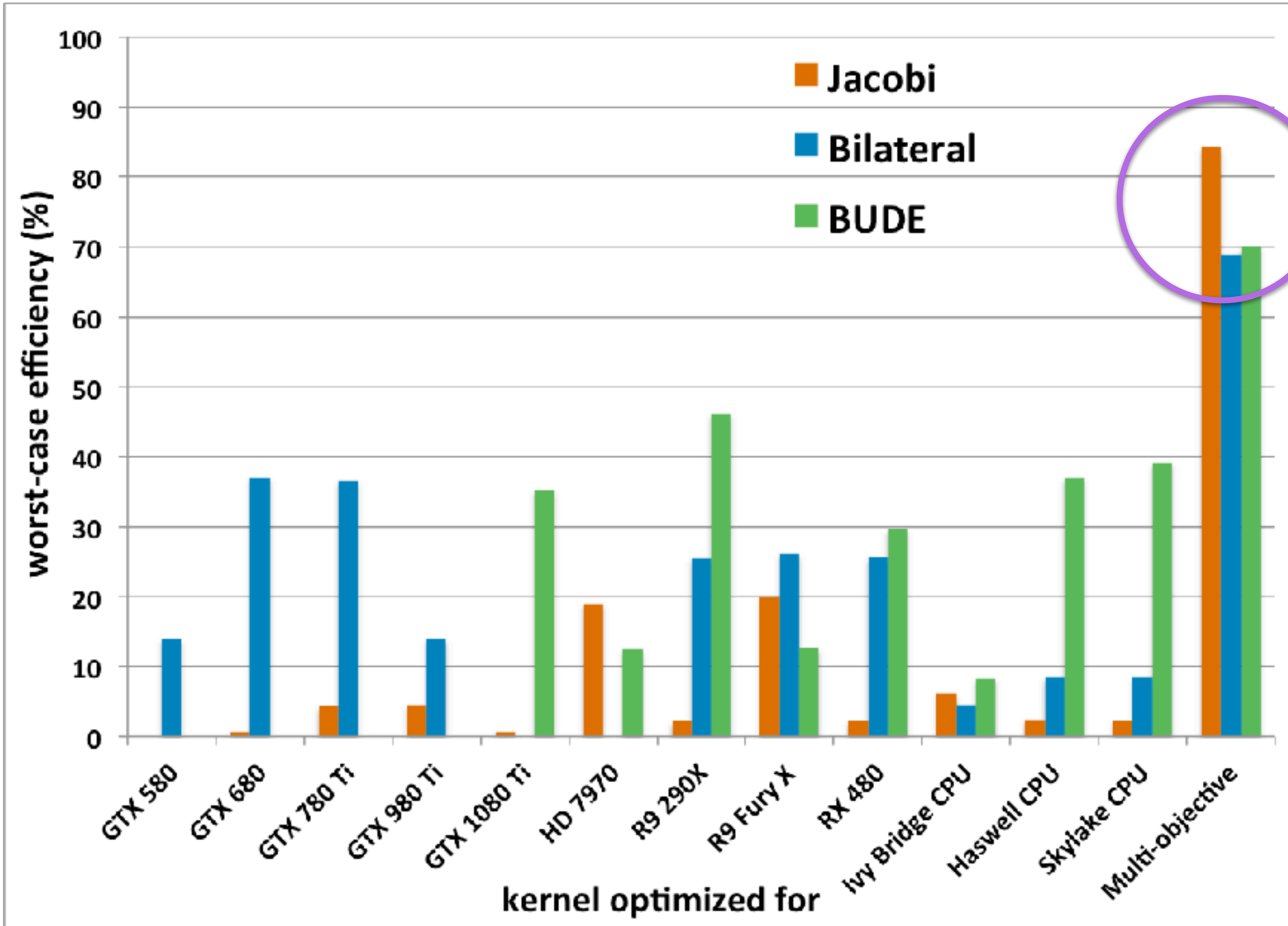
# BUDE efficiencies

| tuned for \ running on | GTX 580 | GTX 680 | GTX 780 Ti | GTX 980 Ti | GTX 1080 Ti | HD 7970 | R9 290X | R9 Fury X | RX 480 | Ivy Bridge CPU | Haswell CPU | Skylake CPU | **WCE** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Skylake CPU | 74% | 64% | 39% | 85% | 87% | 84% | 61% | 47% | 92% | 98% | 98% | 100% | **39%** |
| Haswell CPU | 81% | 81% | 37% | 90% | 93% | 78% | 64% | 48% | 92% | 98% | 100% | 98% | **37%** |
| Ivy Bridge CPU | 15% | 8% | 13% | 21% | 29% | 10% | 10% | 12% | 12% | 100% | 98% | 99% | **8%** |
| RX 480 | 68% | 30% | 32% | 90% | 96% | 72% | 67% | 50% | 100% | 63% | 61% | 57% | **30%** |
| R9 Fury X | 91% | 52% | 32% | 79% | 86% | 72% | 66% | 100% | 100% | 16% | 14% | 13% | **13%** |
| R9 290X | 84% | 50% | 46% | 89% | 97% | 75% | 100% | 86% | 98% | 93% | 93% | 93% | **46%** |
| HD 7970 | 90% | 78% | 33% | 66% | 74% | 100% | 73% | 86% | 95% | 15% | 15% | 12% | **12%** |
| GTX 1080 Ti | 91% | 48% | 35% | 77% | 100% | 71% | 61% | 71% | 75% | 62% | 61% | 57% | **35%** |
| GTX 980 Ti | 91% | 66% | 83% | 100% | 100% | X | X | X | X | X | X | X | **-** |
| GTX 780 Ti | X | 81% | 100% | 89% | 83% | X | X | X | X | X | X | X | **-** |
| GTX 680 | 100% | 100% | X | X | X | X | X | X | X | 57% | 51% | 43% | **-** |
| GTX 580 | 100% | 98% | X | X | X | X | X | X | X | 57% | 51% | 43% | **-** |

# Multi-objective auto-tuning

- Extend tuning process to consider multiple devices at once
- Each time a kernel is generated, the auto-tuner evaluates it on every target device
- The performance values are then reduced into a single number representing the overall ‘fitness’
- We use worst-case efficiency for this fitness function

# Multi-objective tuning results



# Summary

- Over-optimisation hurts performance portability
- Auto-tuning can be a great way to expose these issues
- It can also help generate performance portable kernels
- Future work looking at tuning across different input/problem configurations