Performance Prediction for GPUs and CPUs

James Stevens

Dr. Andreas Klöckner

Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

- Computational kernel
  - (Multi-threaded) shared memory computation
  - Often small piece of larger application that may be offloaded to coprocessor
  - Solve $Ax = b$

Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

- Transformable
  - Kernel algorithm alterable without re-writing kernel code

Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

- Portable
  - Runs reasonably well on various architectures

Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

- Cheap
  - Performance gains outweigh calibration costs

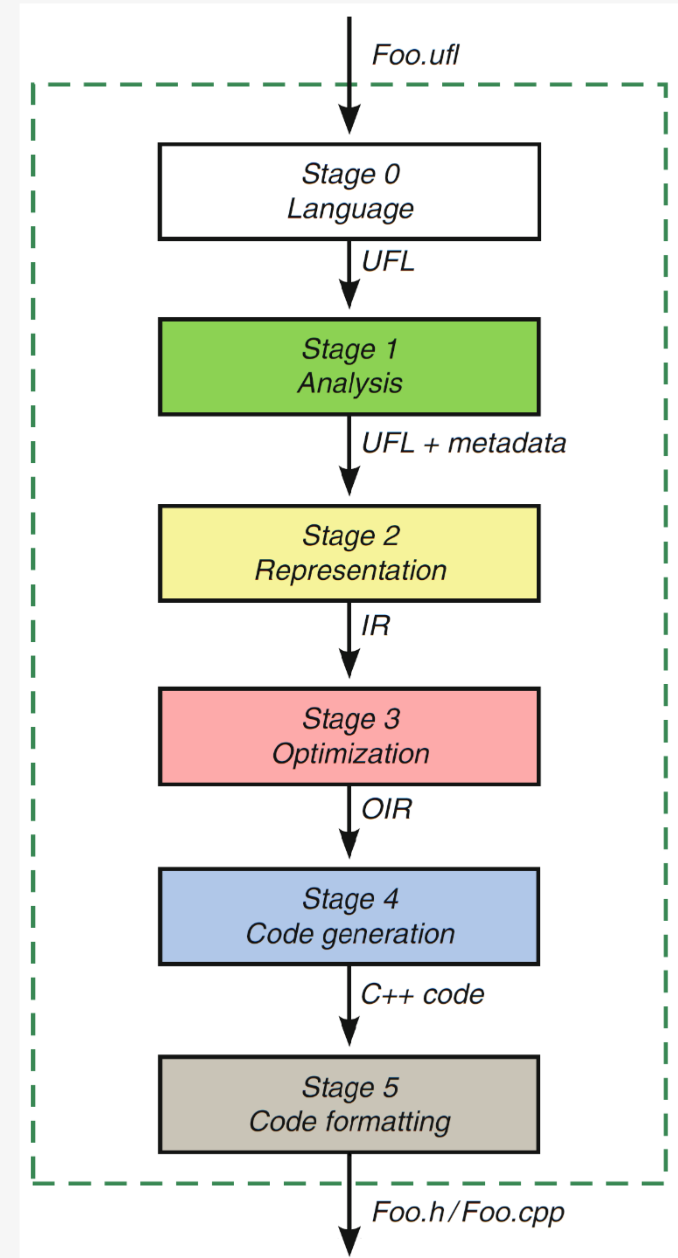
Goal

- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

- Easily
  - Maximizing ratio of time spent writing meaningful instructions to time spent implementing tedious details

Example application

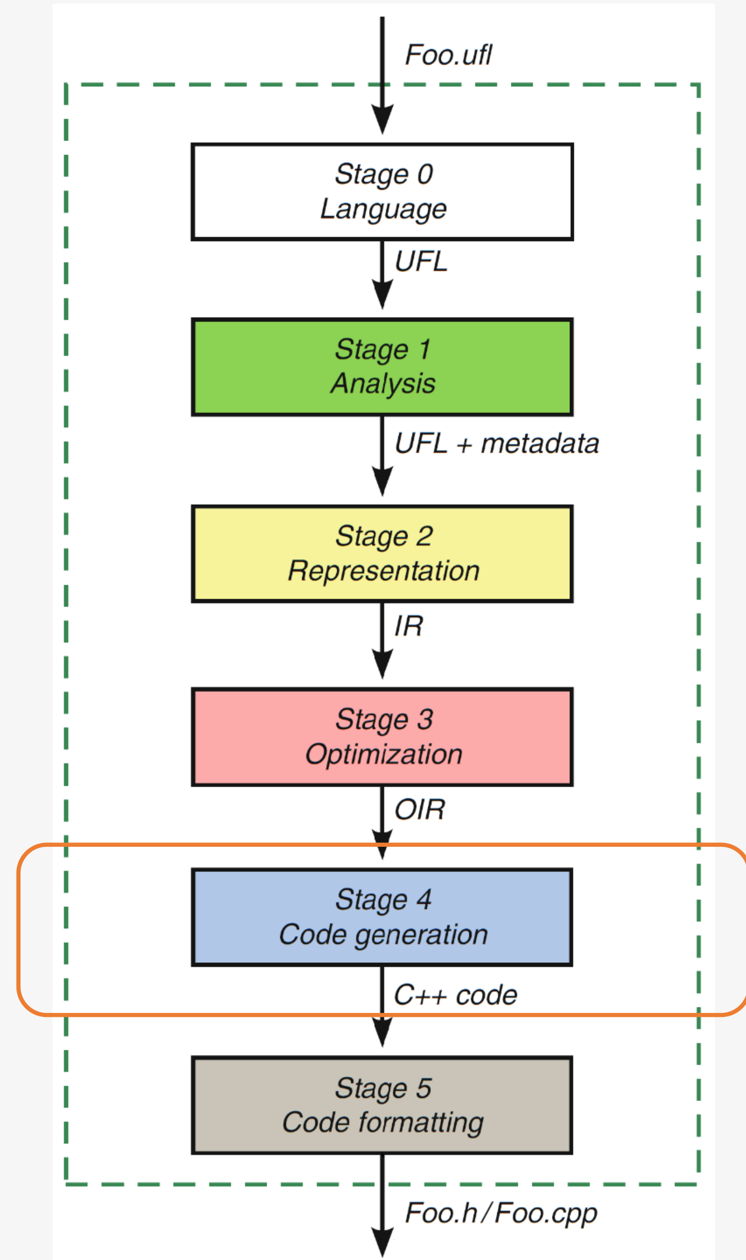
- Automated solution to PDEs
  - FEniCS



Example application

- Automated solution to PDEs
  - FEniCS

Generate optimized code

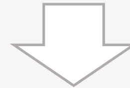


Loo.py

- Programming system embedded in Python providing transformation-based code generation for GPUs and CPUs
- Transformation examples:
  - Loop tiling
  - Vectorization
  - Prefetching
  - Unrolling
  - Split and tag
  - Change data layout

Loo.py kernel generation

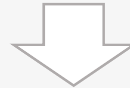
```
knl_0 = lp.make_kernel("{ [i]: 0<=i<n }", "a[i]=b[i]")
```



```
---------------------------------------------------------------------------
KERNEL: loopy_kernel
---------------------------------------------------------------------------
ARGUMENTS:
a: GlobalArg, type: <runtime>, shape: (n), dim_tags: (N0:stride:1)
b: GlobalArg, type: <runtime>, shape: (n), dim_tags: (N0:stride:1)
n: ValueArg, type: <runtime>
---------------------------------------------------------------------------
DOMAINS:
[n] -> { [i] : i >= 0 and i <= -1 + n }
---------------------------------------------------------------------------
INSTRUCTIONS:
[i]                                     a[i] <- b[i]   # insn
---------------------------------------------------------------------------
```

Loo.py code generation

```
knl_0 = lp.make_kernel("{ [i]: 0<=i<n }", "a[i]=b[i]")
```



OpenCL kernel:

```
__kernel void __attribute__ ((reqd_work_group_size(1,1,1)))
loopy_kernel( < ...parameters... > )
{
  for (int i = 0; i <= -1 + n; ++i)
    a[i] = b[i];
}
```

Loo.py transformations

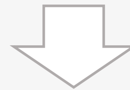
- Example: split index to distribute work
  - Parallelize loop with thread blocks of size 128

```
knl_1 = lp.split_iname(knl_0,"i",128
                       ,outer_tag="g.0",inner_tag="l.0")
```

Loo.py transformations

Original kernel:

```
__kernel void __attribute__ ((reqd_work_group_size(1,1,1)))
loopy_kernel( < ...parameters... > )
{
  for (int i = 0; i <= -1 + n; ++i)
    a[i] = b[i];
}
```



Transformed kernel:

```
__kernel void __attribute__ ((reqd_work_group_size(128,1,1)))
loopy_kernel( < ...parameters... > )
{
  if (-1 + -128 * gid(0) + -1 * lid(0) + n >= 0)
    a[lid(0) + gid(0) * 128] = b[lid(0) + gid(0) * 128];
}
```

Loo.py transformations

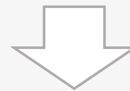
- Example: loop unrolling
  - Unroll original loop with step size 4
  - Need assumptions

```
knl_0 = lp.make_kernel("{ [i]: 0<=i<n }", "a[i]=b[i]",
                            assumptions="n>=0 and n mod 4 = 0")
knl_2 = lp.split_iname(knl_0,"i",4)
knl_2 = lp.tag_inames(knl_2,dict(i_inner="unr"))
```

Loo.py transformations

Original kernel:

```
{
  for (int i = 0; i <= -1 + n; ++i)
    a[i] = b[i];
}
```



Transformed kernel:

```
{
  for (int i_outer = 0; i_outer <= -1+((3+n)/4); ++i_outer)
  {
    a[0 + i_outer * 4] = b[0 + i_outer * 4];
    a[1 + i_outer * 4] = b[1 + i_outer * 4];
    a[2 + i_outer * 4] = b[2 + i_outer * 4];
    a[3 + i_outer * 4] = b[3 + i_outer * 4];
  }
}
```

Goal

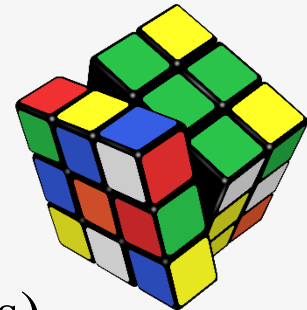
- Easily write portable, transformable computational kernels with cheap, automated calibration for optimal performance

☑ Portable, transformable computational kernels
  - Loo.py allows this

☐ Cheap, automated calibration

Automated Calibration

- Want automated selection of best parameter values and kernel transformations for available hardware
- Evaluate kernel performance empirically?
  - Search space explodes
    - 5 potential transformations, 2 parameters w/ 4 possible values each = 512 configurations
  - Kernel execution = O(seconds)
  - Prediction model evaluation = O(milliseconds)



Automated Calibration

- Need mathematical model that predicts performance

Automated Calibration

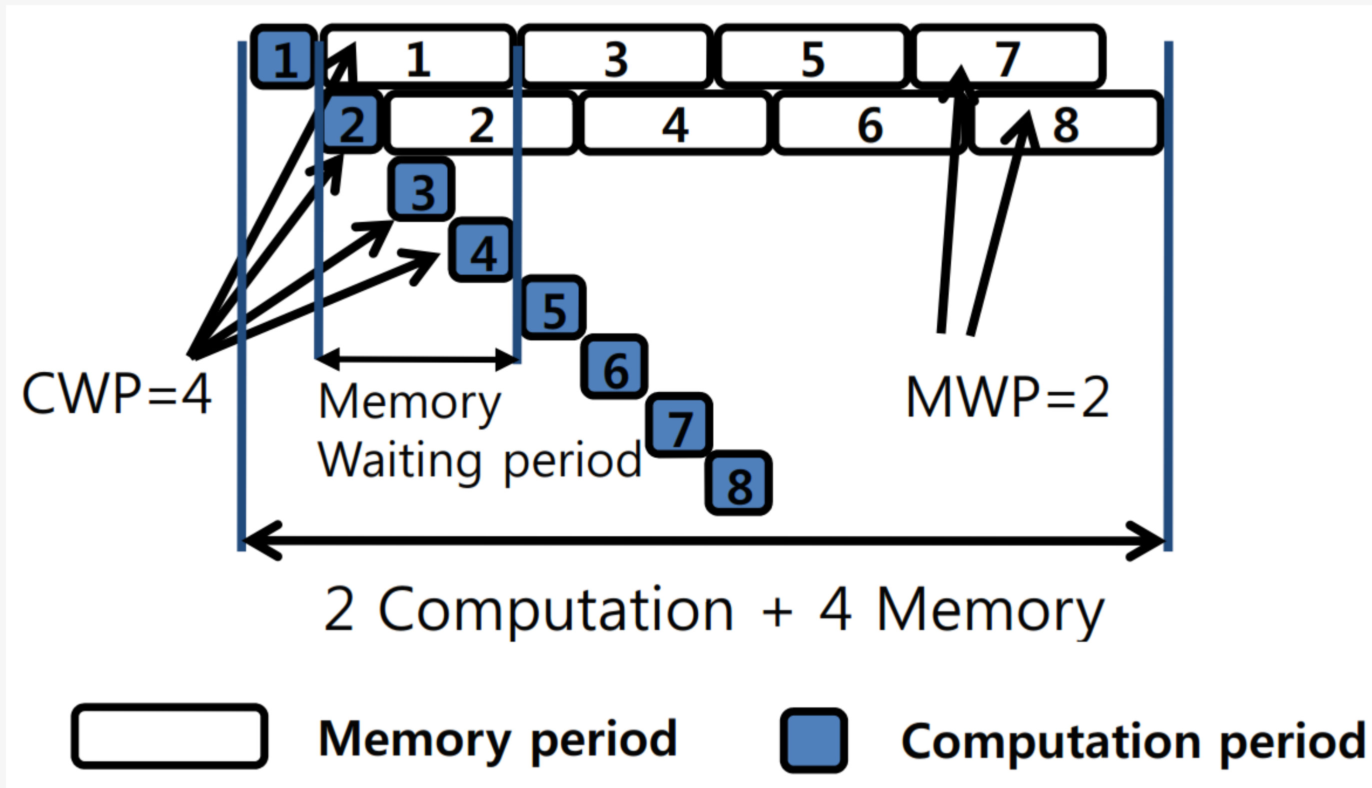
- Need mathematical model that predicts performance

- Hong and Kim, 2009
  - *An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness*

Hong-Kim Performance Model

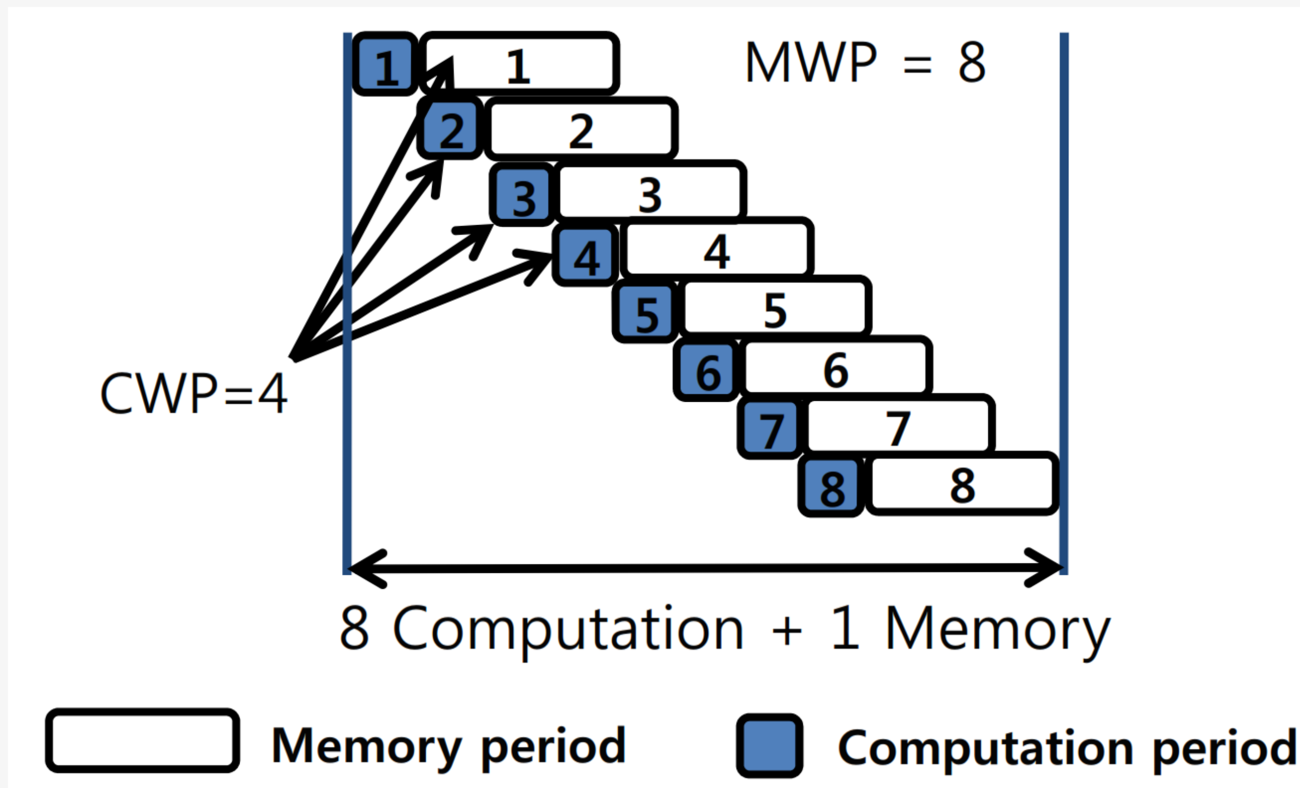
- GPU can execute multiple time-sharing warps (batches of threads) while other warps wait for data from memory
- Two key values introduced
  - Memory warp parallelism (MWP)
    - Number of memory requests that can execute concurrently
    - Determined by amount of memory parallelism in system
      - Mem. bandwidth, mem. bank parallelism, # active warps per SM
  - Computation warp parallelism (CWP)
    - Amount of computation that can occur while one warp waits for memory values
    - Determined by algorithm characteristics

Example: CWP > MWP



- Runtime dominated by memory access

Example: MWP > CWP



- Runtime dominated by computation

Hong-Kim Model Parameters

Hardware

- Threads per warp
- Cycles per instruction
- Clock frequency
- DRAM latency/bandwidth
- Delay between uncoalesced/coalesced memory transactions

+ more

Algorithm

- Computation instructions per thread
- Uncoalesced/coalesced memory instructions per thread
- Synchronization instructions per thread

+ more

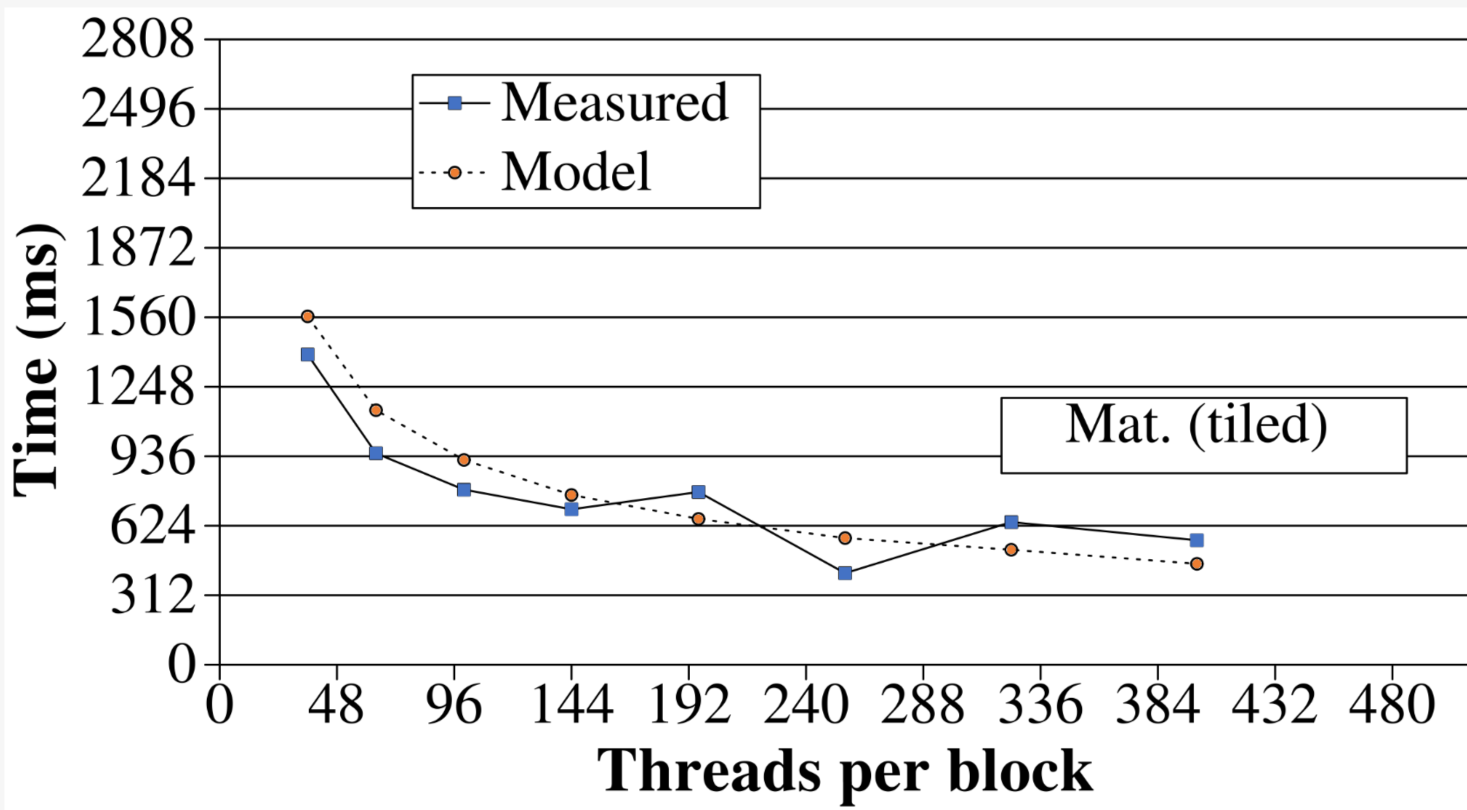
Both

- Active thread blocks per SM

+ more

Hong-Kim Model Accuracy

- Tiled mat-mul (NVIDIA Quadro FX 5600)



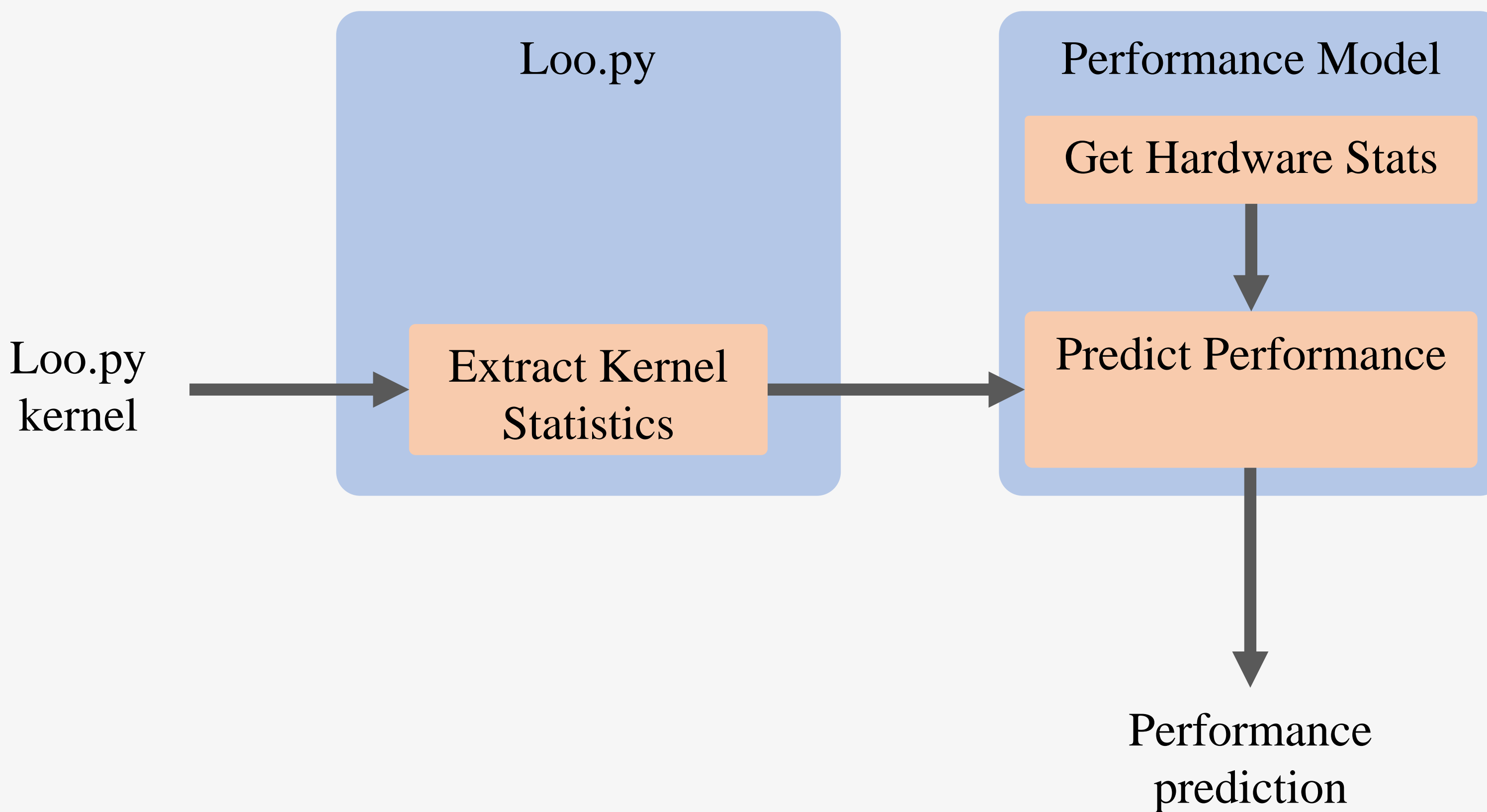
Hong-Kim Model Accuracy

- Report 13.3% avg. error on following benchmarks
  - Sepia – filter for artificially aging images
  - Linear – image filter computing avg. of 9 pixels
  - SVM – kernel for SVM-based algorithm
  - Mat-mul naïve
  - Mat-mul tiled
  - Blackscholes – option pricing algorithm

Initial Implementation

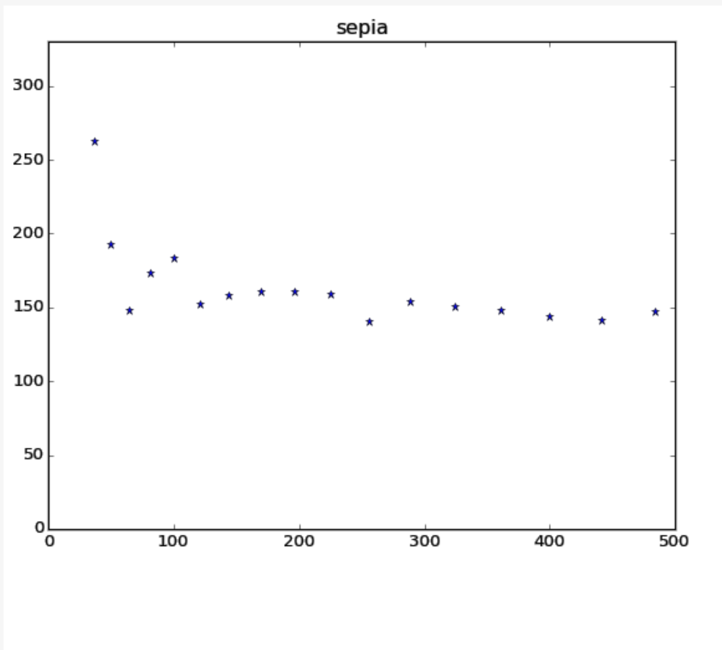
- Functions gather statistics from Loo.py kernel
    - 32-bit flops, coalesced/uncoalesced mem. transactions, barriers
- Class for hardware stats, set values manually
- Feed this information to performance model

Implementation

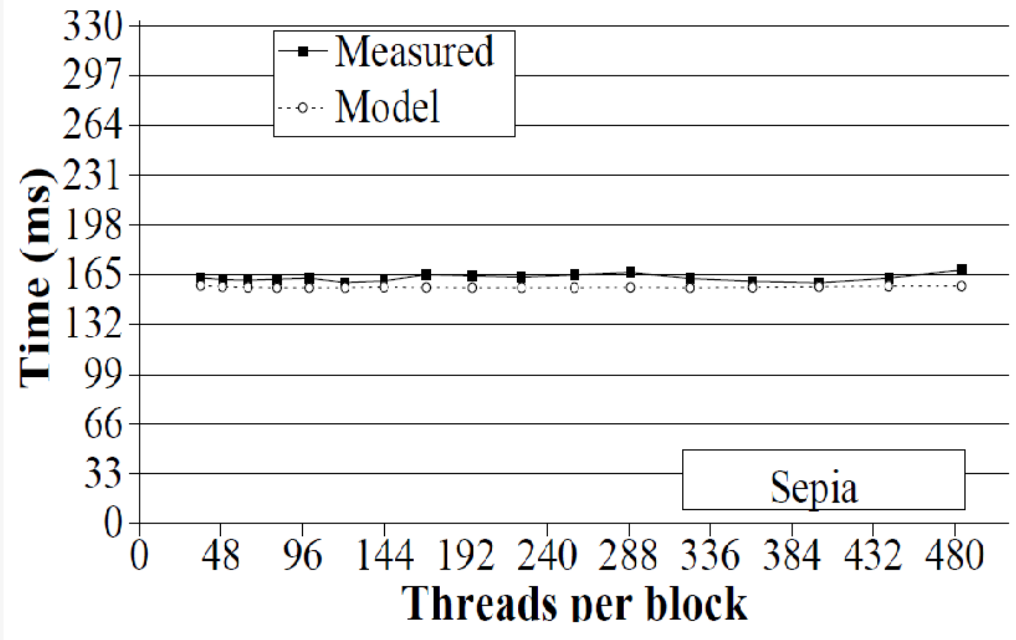


Reproducing H-K Model Output

Output



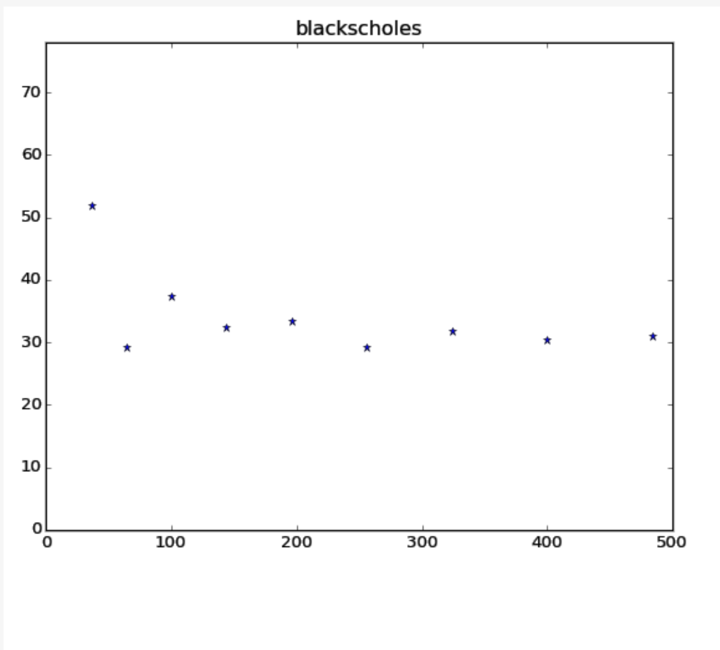
Corresponding plot from HK paper

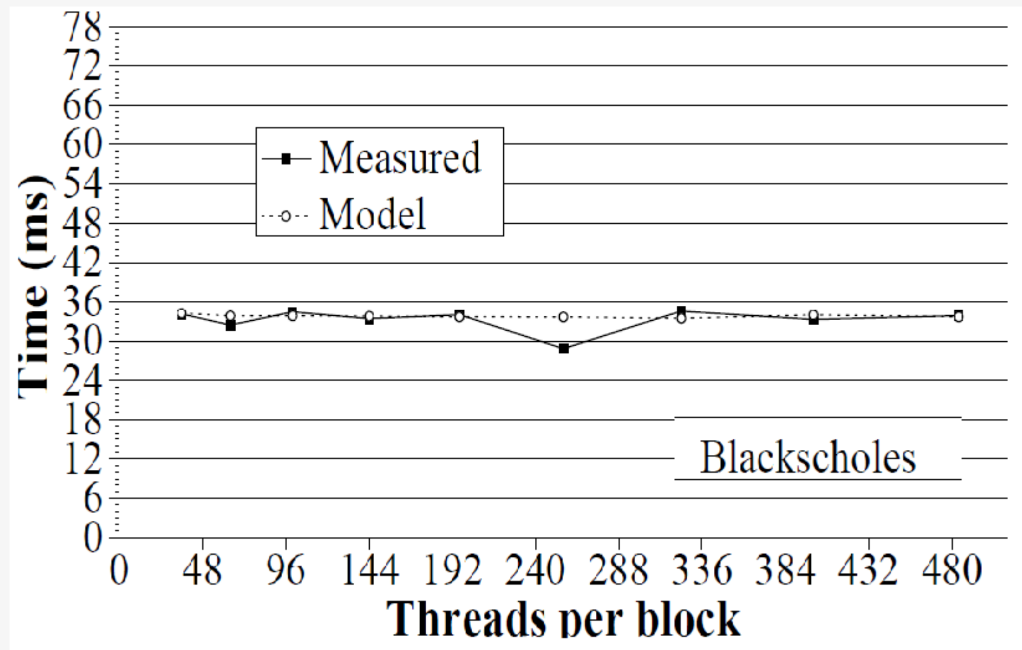


Reproducing H-K Model Output

Output

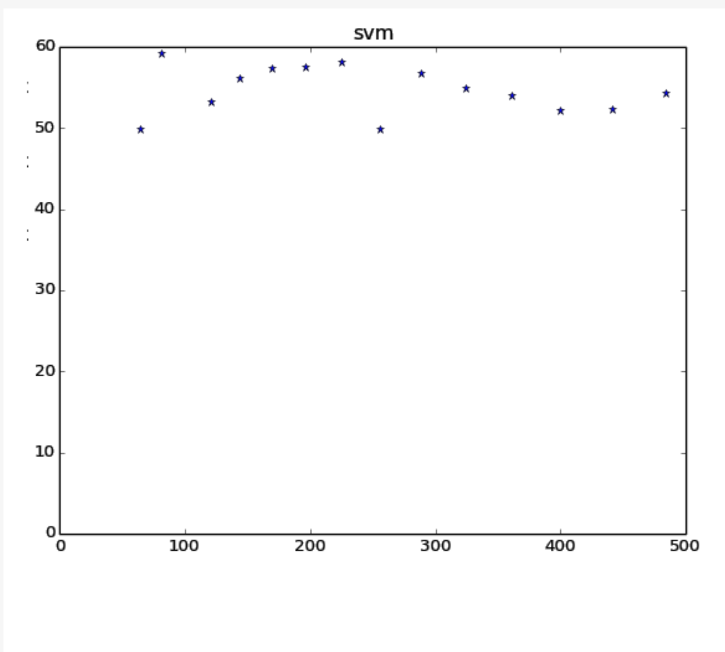
Corresponding plot from HK paper



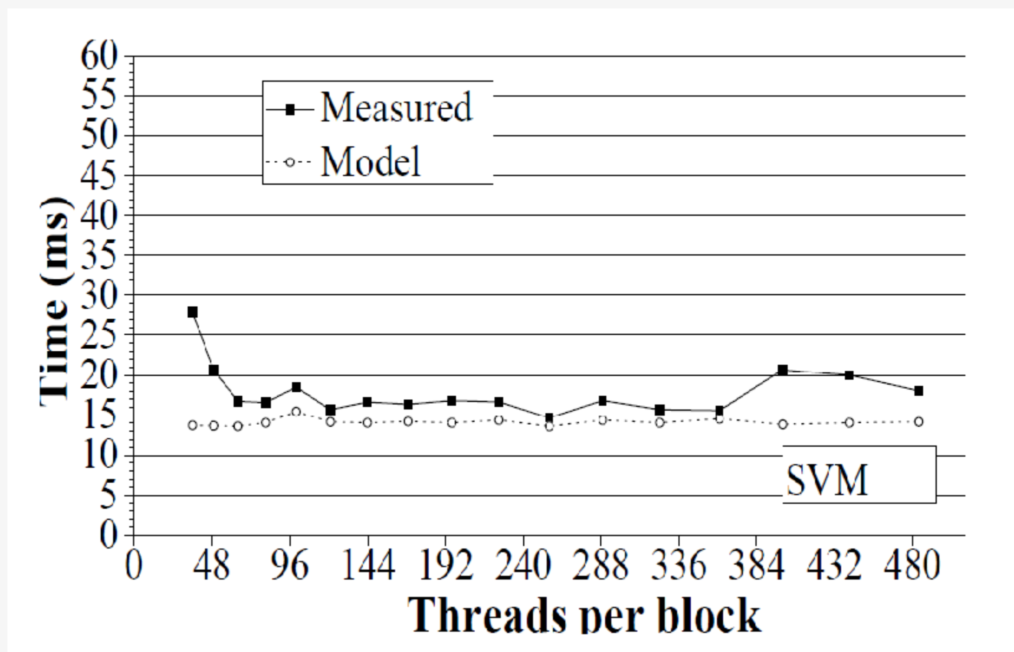


Reproducing H-K Model Output

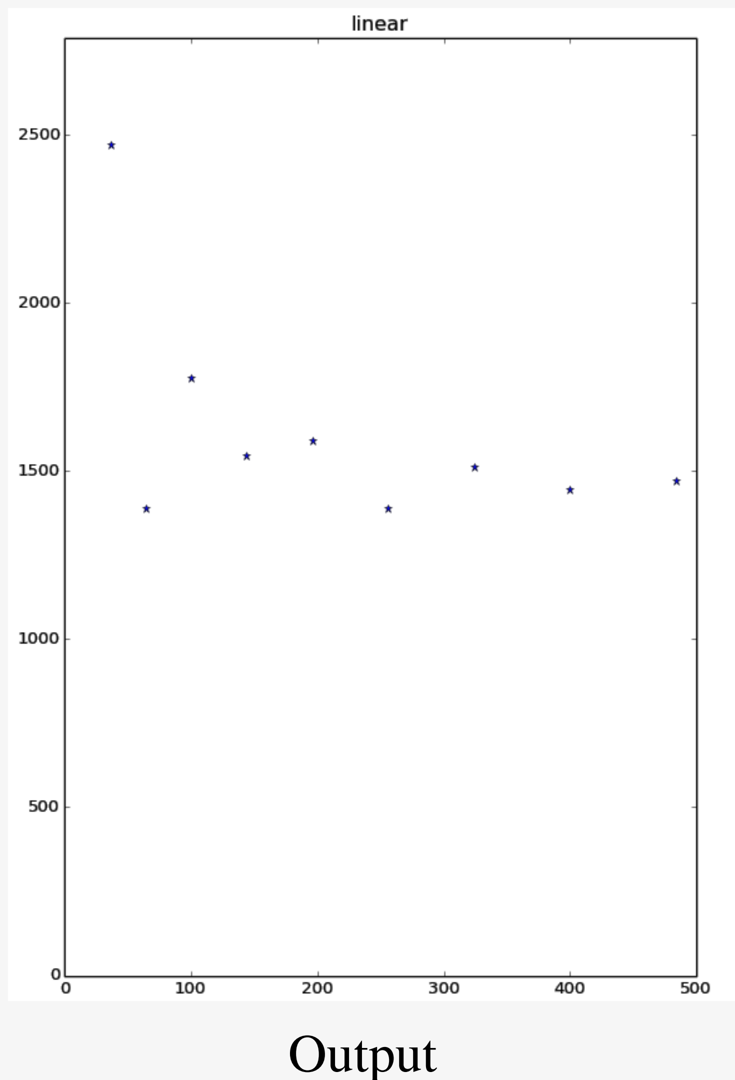
Output



Corresponding plot from HK paper

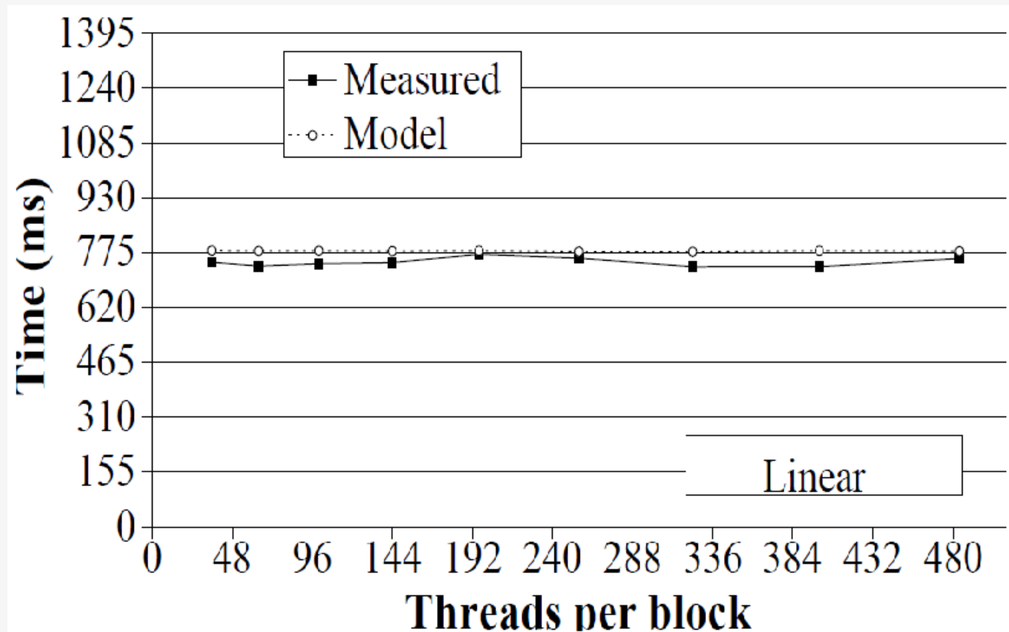


Reproducing H-K Model Output



Output

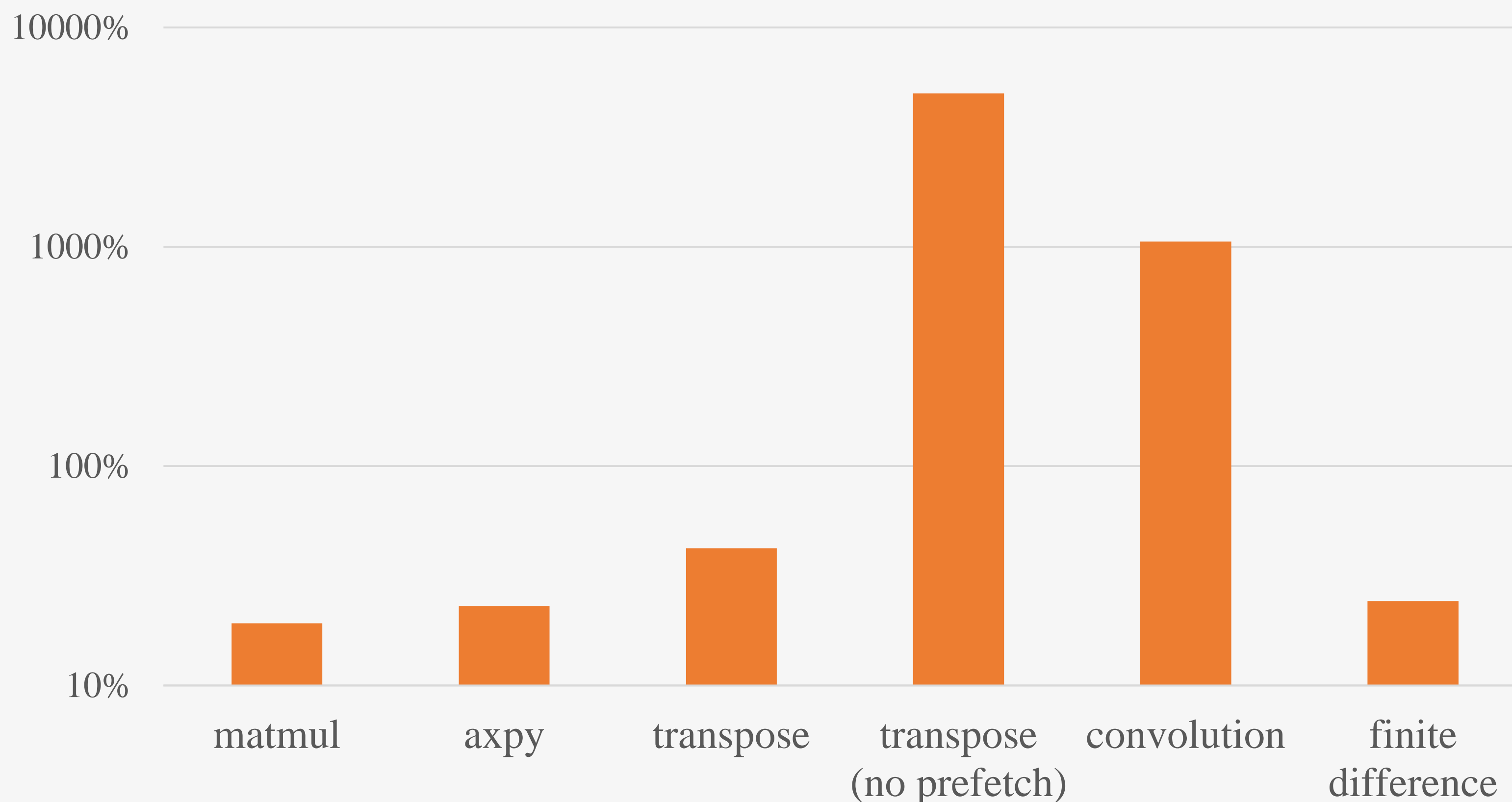
Corresponding plot from HK paper



Reproducing H-K Model Output

- What went wrong with kernels 3 and 4?
  - Same GPU stats (reported in paper)
  - Used same kernel stats (reported in paper), except some parameters not reported…
- Problem: what counts as computation/memory instruction?
  - H-K analyzes assembly instructions
  - We look at flops and fetches in kernel

H-K Prediction Results



H-K Prediction Results

- Significant prediction accuracy variation
  - 10% - 5000% error for various kernels
  - In-exact counting of “computation instructions” and “memory instructions”
  - Newer GPUs behave differently
    - Caching

Difficulties With Original Plan

- Acquiring hardware info
  - 30+, often illusive, values needed
  - Force users to find and enter these manually?
  - Maintain remotely accessible database updated by users?
- Acquiring kernel info
  - Accurate values require analyzing assembly code
    - Slow
    - Difficult to automate
- Hong-Kim model limitations
  - Only for GPUs, only tested on NVIDIA
  - Does not consider caching, control flow divergence

Difficulties With Original Plan

- Hong-Kim and similar models rely on analysis that is difficult to automate, and are tailored for specific kinds of hardware
- Predict performance without building model that depends on difficult to gather information?

Idea: Linear Least Squares

- Can we predict run time as a linear combination of kernel statistics?
  - When installing on new system, run set of calibration kernels and use linear least squares to produce weights for each kernel characteristic
  - Performance prediction is just vector inner-product

Linear Least Squares

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \cdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_m \end{bmatrix} = b$$

Linear Least Squares

$$Ax = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \cdots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_m \end{bmatrix} = b$$

- Each row of A
  - Kernel stats (e.g. barrier, operation, and fetch counts)
  - Entered on per-thread basis, multiplied by total batches of threads executed

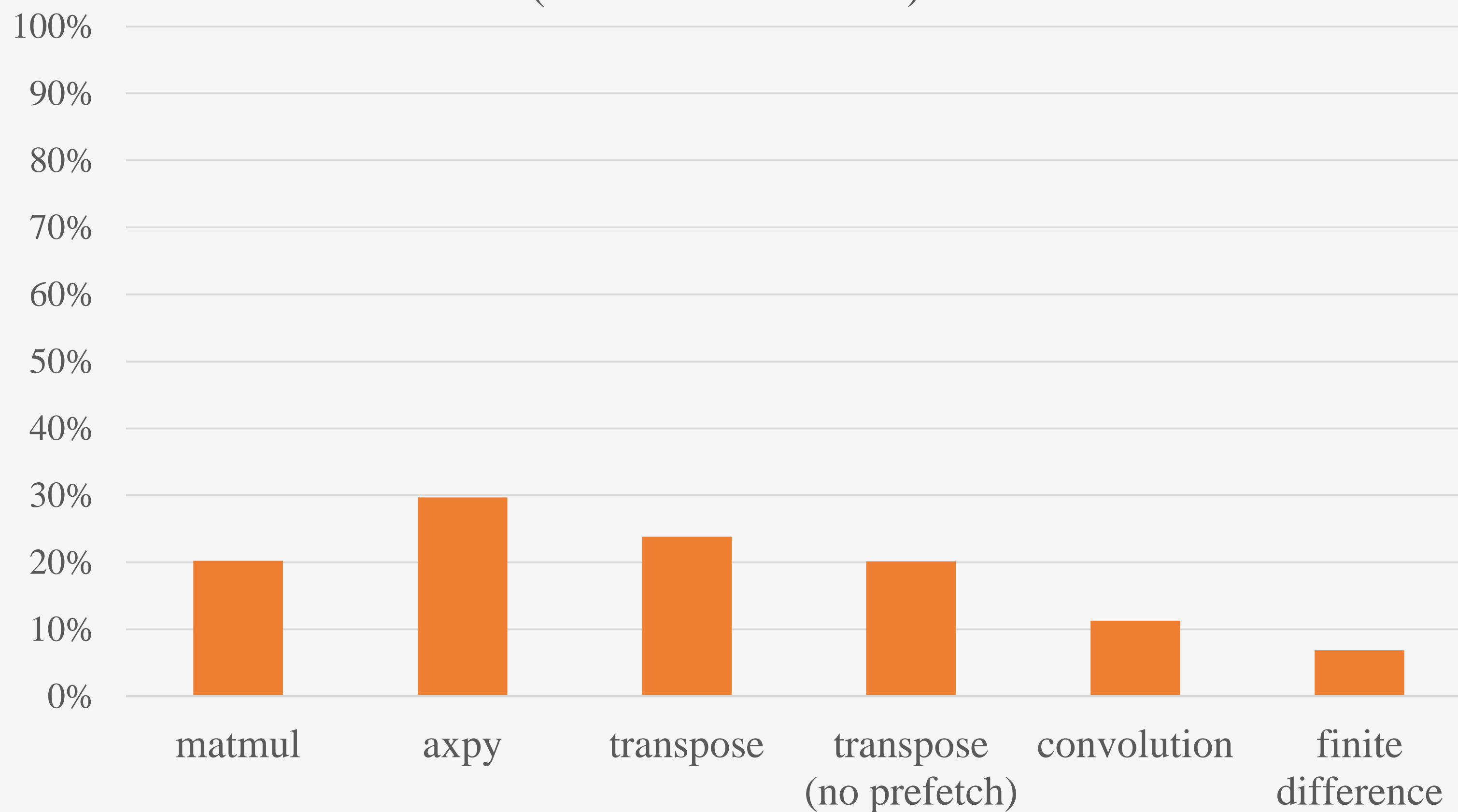
Linear Least Squares

- Gather more specific kernel stats
  - Comp. instructions → $\pm$, $\times$, $\div$, pow, bitwise, fmad
  - Mem. instructions → store, fetch, min(store, fetch)
  - All comp. and mem. Instructions categorized by data size
    - Int, float32, float64

Linear Least Squares

- Initial test: can we predict performance for kernels similar to least squares training set?
  - Yes

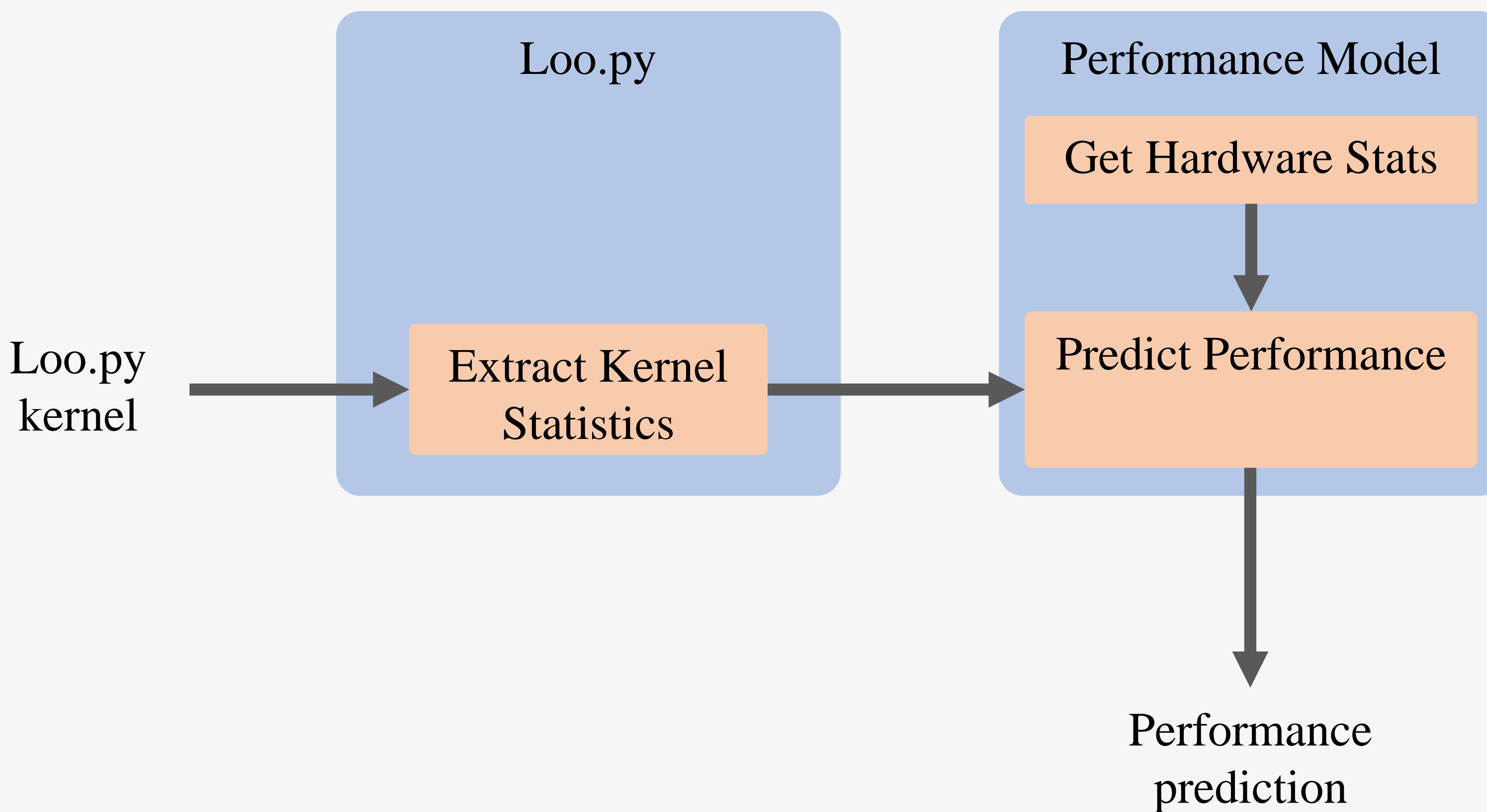
Linear Least Squares



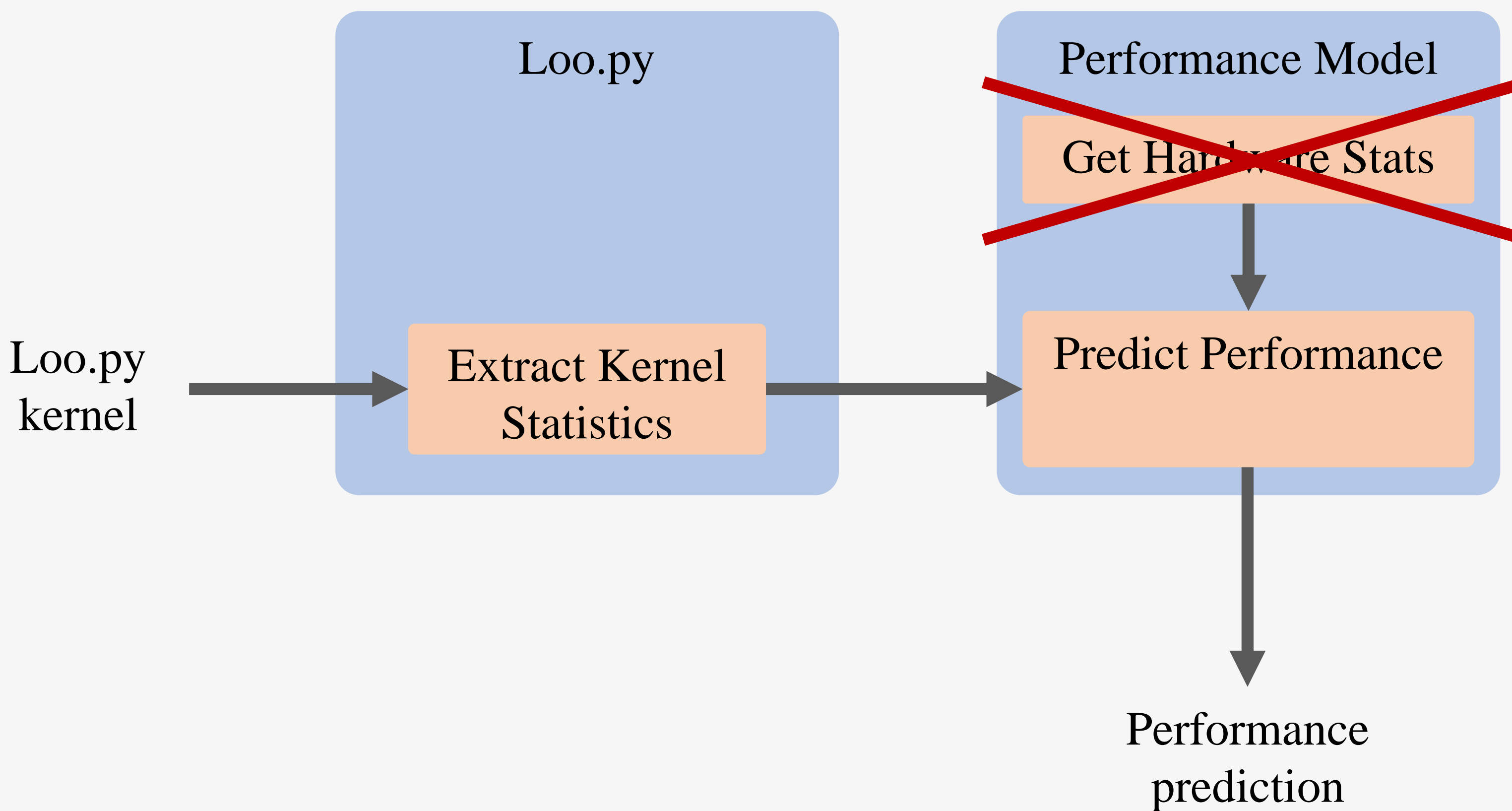
Linear Least Squares

- Can we predict performance for arbitrary kernels with some fundamental training set?
  - Currently working on this
- Initial results suggest that we can
  - Training set with three kinds of kernels
    - Vary op count and keep everything else constant
    - Vary fetch count and keep everything else constant
    - Empty kernel
    - All with varying thread configurations
  - Average rel. error magnitude **24%** when predicting various finite difference kernel configurations

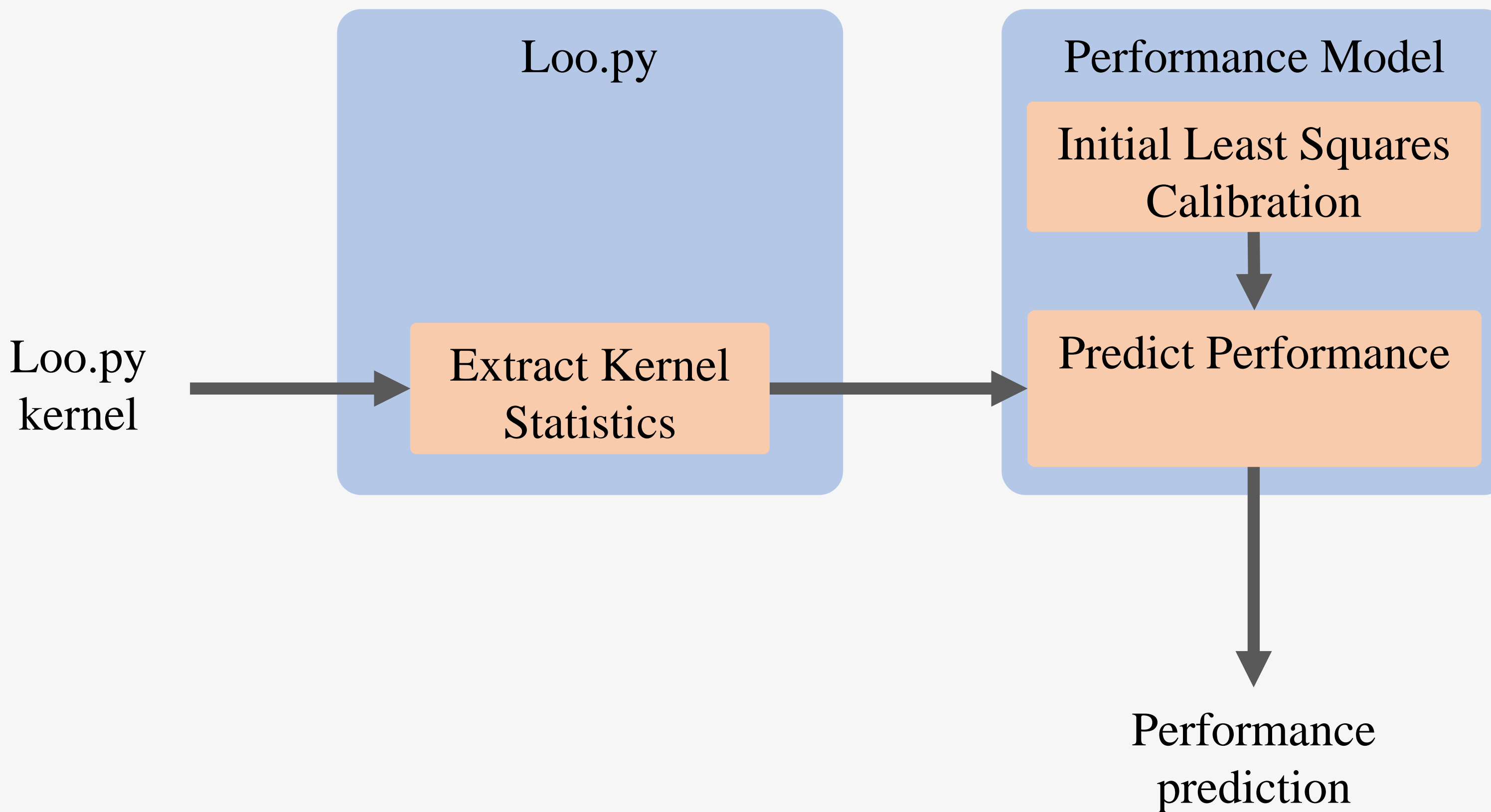
Original plan



New plan



New Plan



New Plan

- Predicted benefits over Hong-Kim and similar models
  - Inputs can be gathered automatically, thanks to Loo.py
  - Fast model evaluation
  - More adaptable to changing hardware

What's next?

- Finish building and evaluating calibration kernel set
- Test on non-GPU architectures

- Upcoming challenges
  - Compilers/schedulers optimizing without permission
    - Data reuse
  - Accuracy on non-GPUs

Questions?

Photo sources

http://www.intel.com/content/dam/www/public/us/en/images/product/RWD/xeon-phi-passive-angle-rwd.png/_jcr_content/renditions/intel.web.256.144.png

http://www.velocitymicro.com/images/upload/tesla-k40-3qtr.png

http://www.amd.com/PublishingImages/photography/product/360px/amd-firepro-s9170-flatangle.png

http://wikigrewal.com/wp-content/uploads/2013/08/xeon_processor.png

http://www.amd.com/PublishingImages/photography/product/360px/amd-opteron-4000.png

FEniCS book

https://upload.wikimedia.org/wikipedia/commons/thumb/a/a6/Rubik's_cube.svg/2000px-Rubik's_cube.svg.png

Hong-Kim Performance Model

$$MWP = \min\left(BWperWarp, \frac{memBW}{BWperWarp \times ActiveSMs}, N\right)$$

$$CWP = \min\left(\frac{cycles_{mem} + cycles_{comp}}{cycles_{comp}}, N\right)$$

$$BWperWarp = \frac{freq \times warpLB}{memL}$$

$warpLB$ = load bytes per warp
$memL$ = round trip time to DRAM
$memBW$ = memory bandwidth
$cycles_{mem}$ = memory waiting cycles per warp
$cycles_{comp}$ = computation cycles per warp
N = active warps per multiprocssor