Performance Prediction for GPUs and CPUs

James Stevens

Dr. Andreas Klöckner

Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

TBD

- Roadmap

Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

} TBD

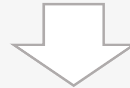
- Roadmap

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

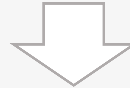
```
knl_0 = lp.make_kernel("{ [i]: 0<=i<n }", "a[i]=b[i]")
```



```
---------------------------------------------------------------------------
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

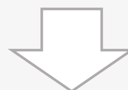
- Example: split index to distribute work
  - Parallelize loop with thread blocks of size 128

```
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

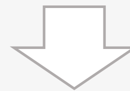
- Example: loop unrolling
  - Unroll original loop with step size 4
  - Need assumptions

```
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];
  }
}
```

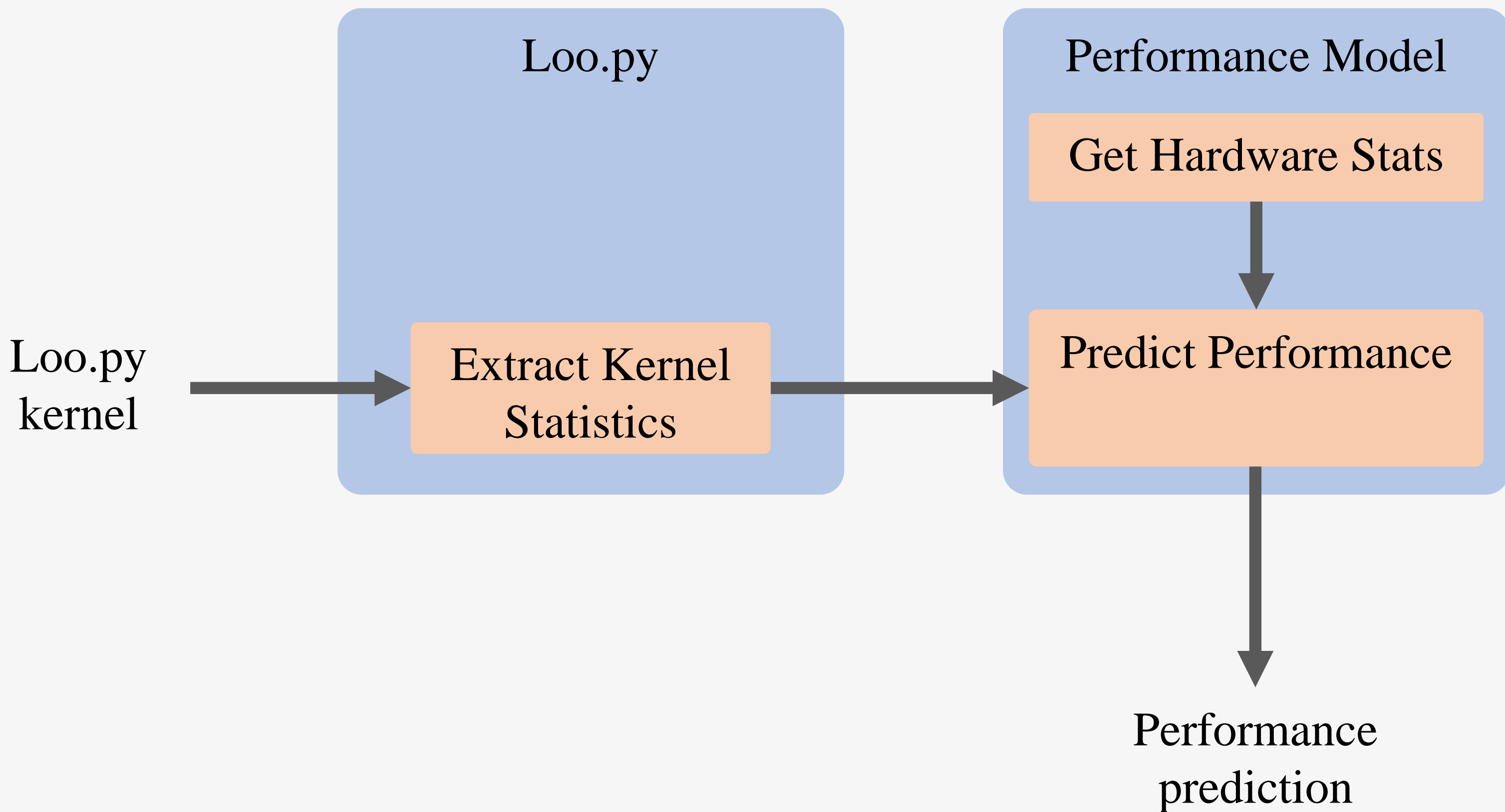
Goal

- Easily write portable transformable computational kernels with low calibration cost for optimal performance
    - Current state: Loo.py allows writing of transformable computational kernels
    - Need: low-cost calibration for optimal performance
        - Performance model

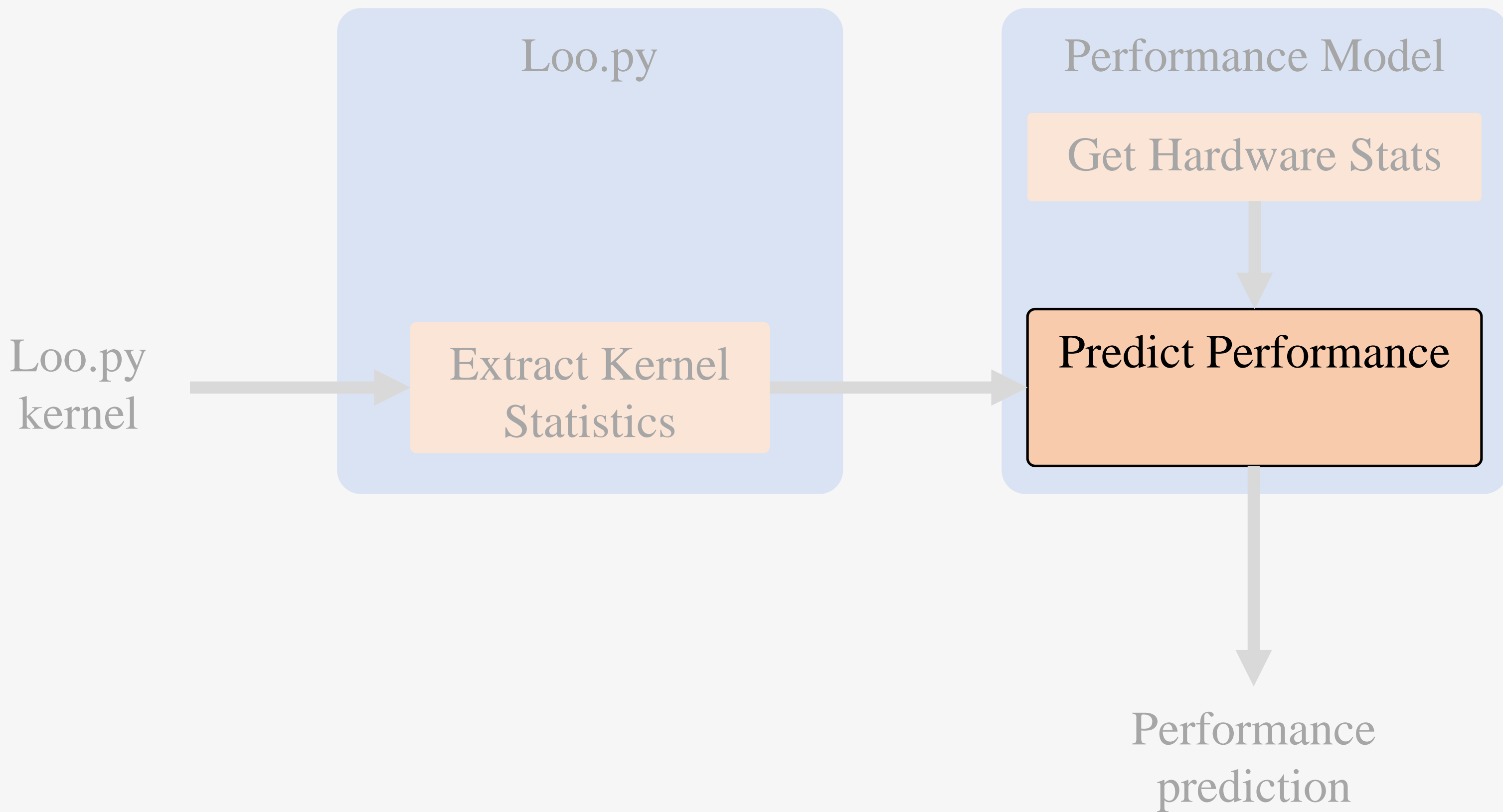
Goal

- Create model that predicts performance of Loo.py kernel on given hardware
  - Allows user to write code that automatically selects best set of kernel transformations and parameters for available hardware
- Why not evaluate kernel performance empirically?
  - 5 potential transformations and 2 parameters with 4 possible values each = 512 kernel configurations
  - Kernel execution = O(seconds)
  - Prediction model evaluation = O(milliseconds)

Integration with Loo.py



Integration with Loo.py



Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

} TBD

- Roadmap

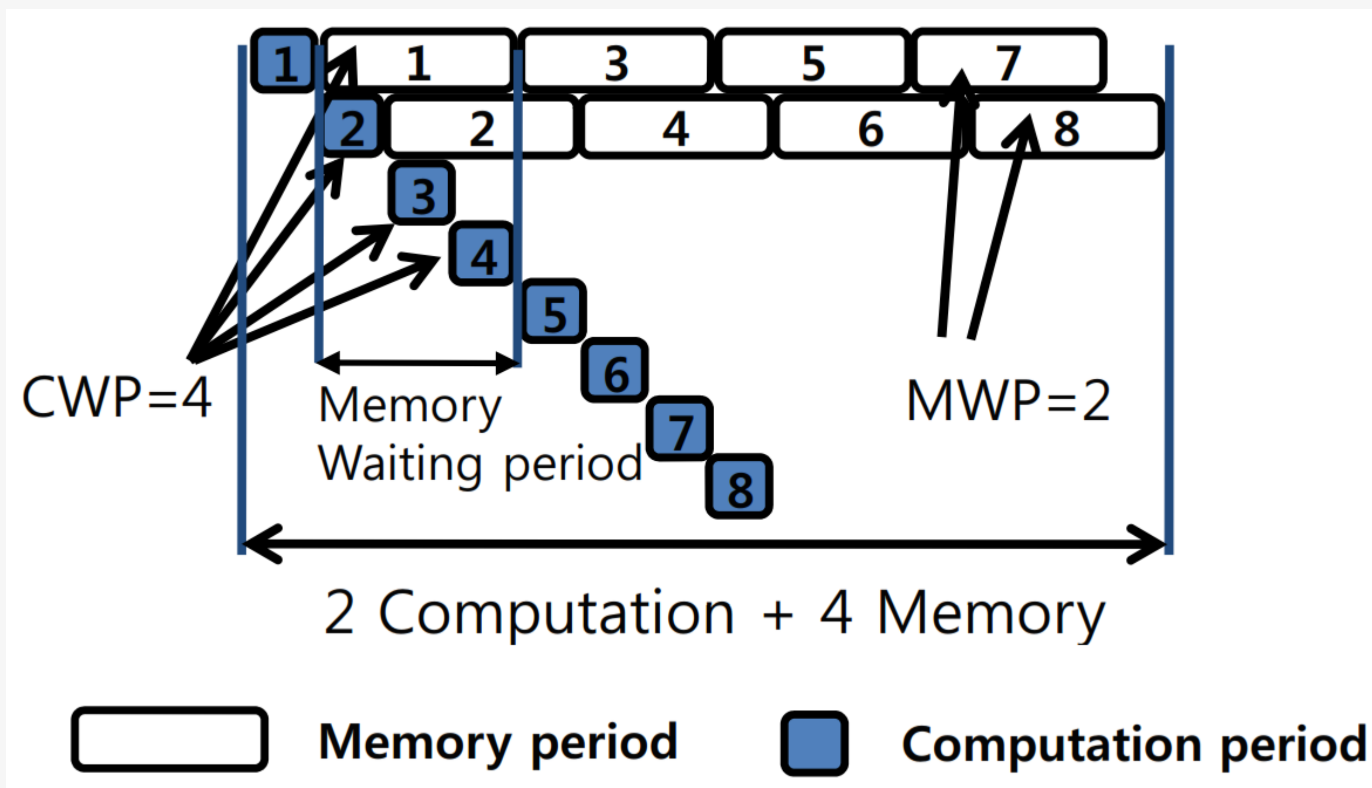
Performance Model Starting Point

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

Hong-Kim Performance Model

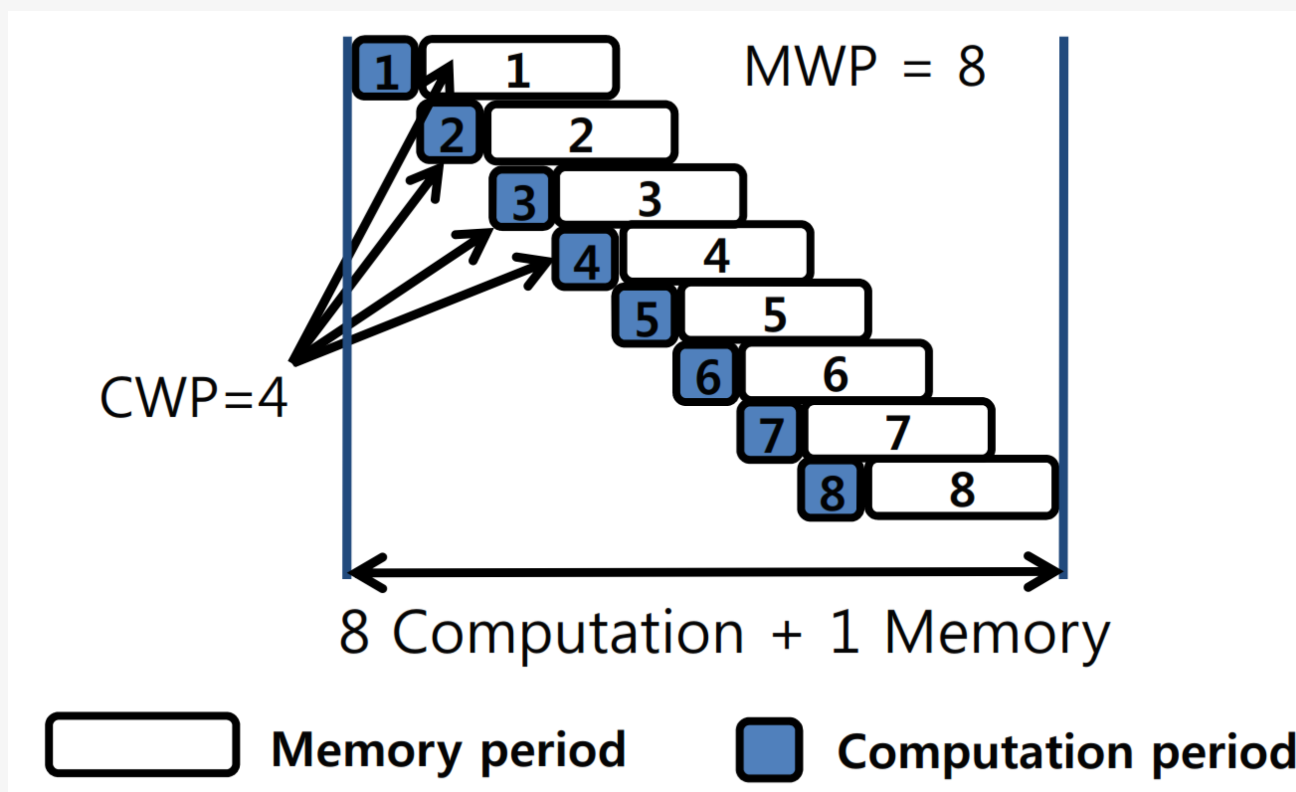
- GPU can execute multiple time-sharing warps (batches of threads) while other warps wait for data from memory
- Two key values introduced
  - Memory warp parallelism (MWP)
    - Number of memory requests that can execute concurrently
    - Determined by amount of memory parallelism in system
      - 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

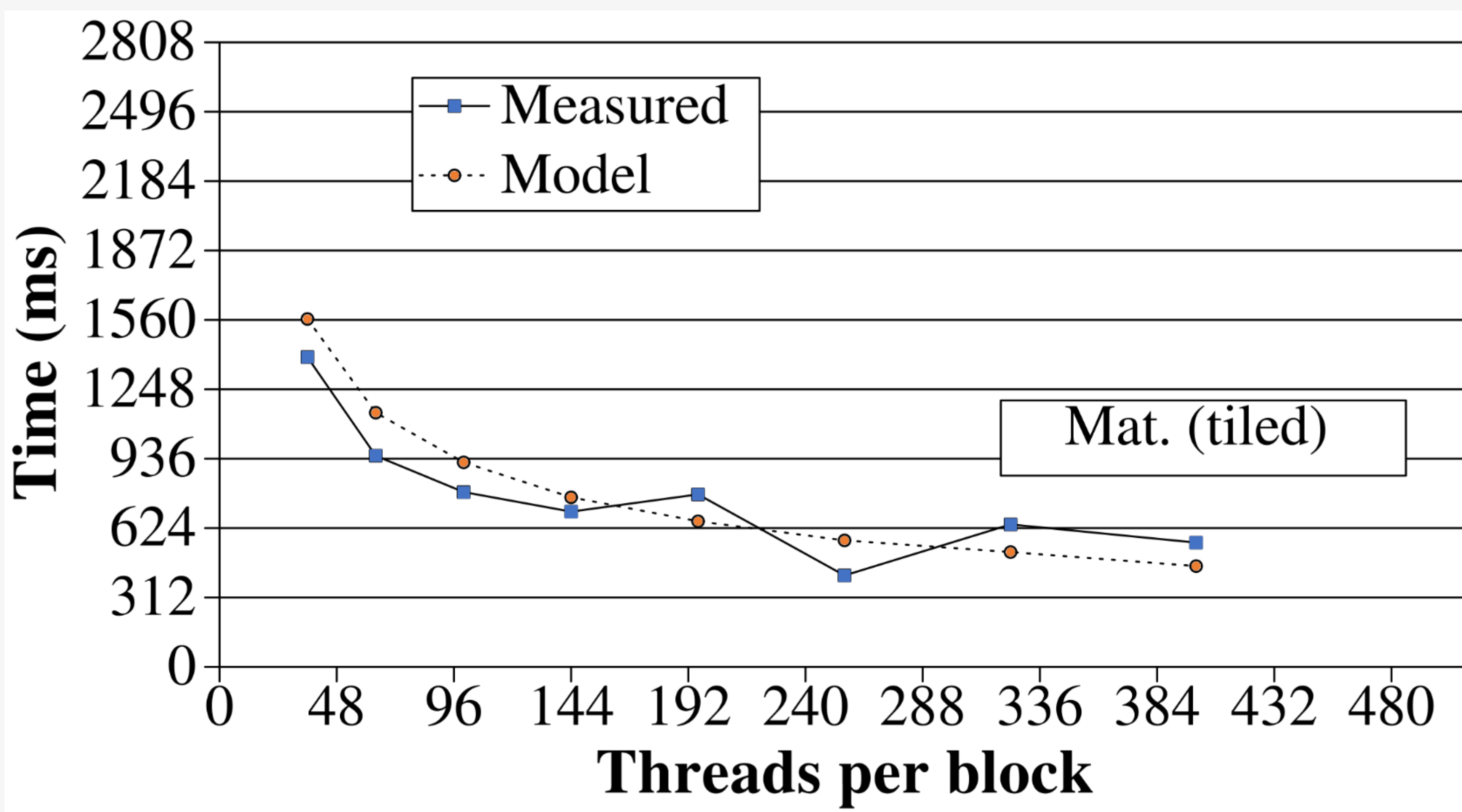
Both

- 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 using 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

Hong-Kim Model Limitations

- Does not consider
  - Control flow divergence
  - Memory bank conflicts
  - Constant, texture cache misses
- Tested only with CUDA (four NVIDIA GPUs)
- Tested only with single-precision data
- Not for multi-core CPUs

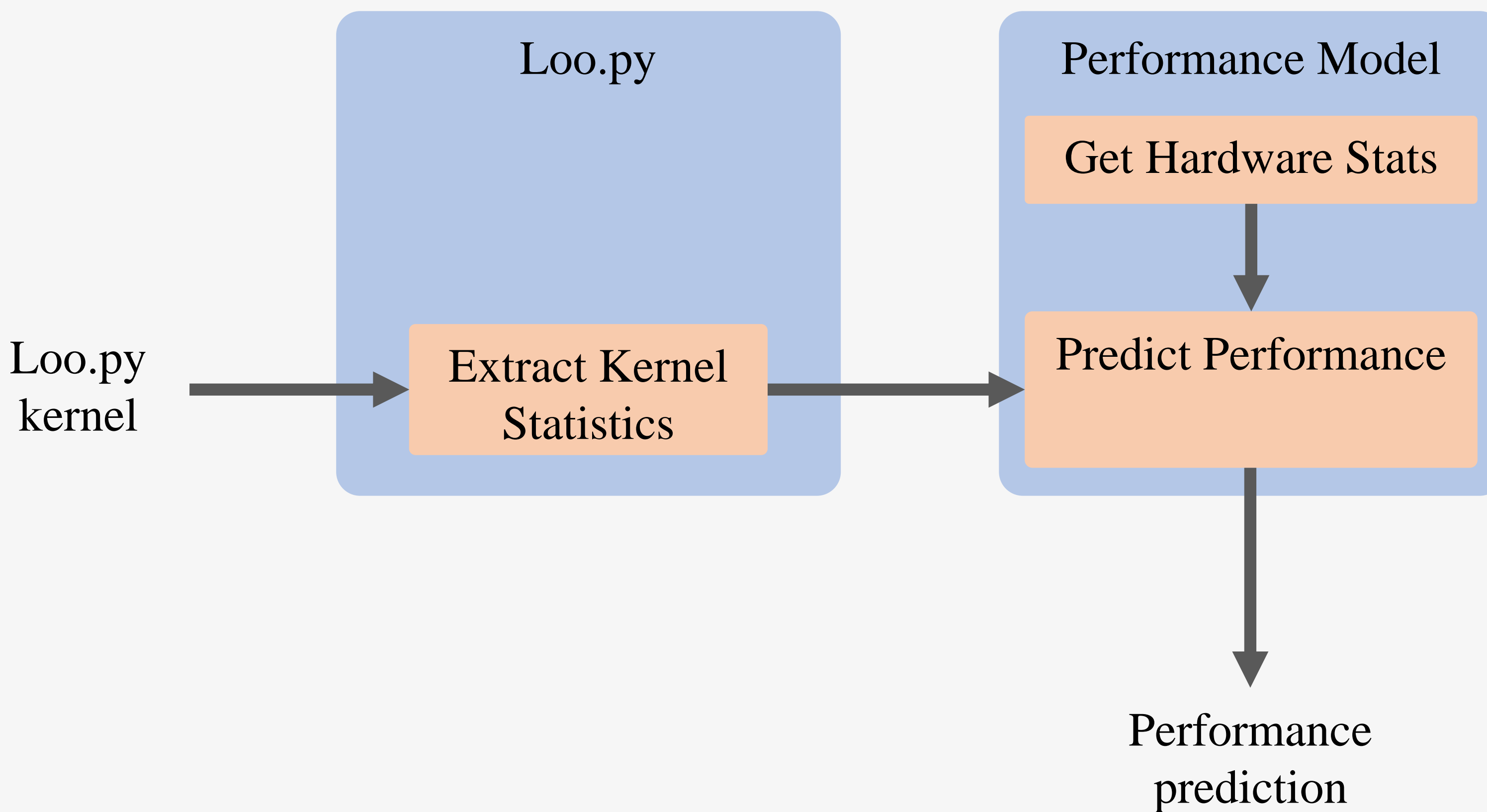
Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

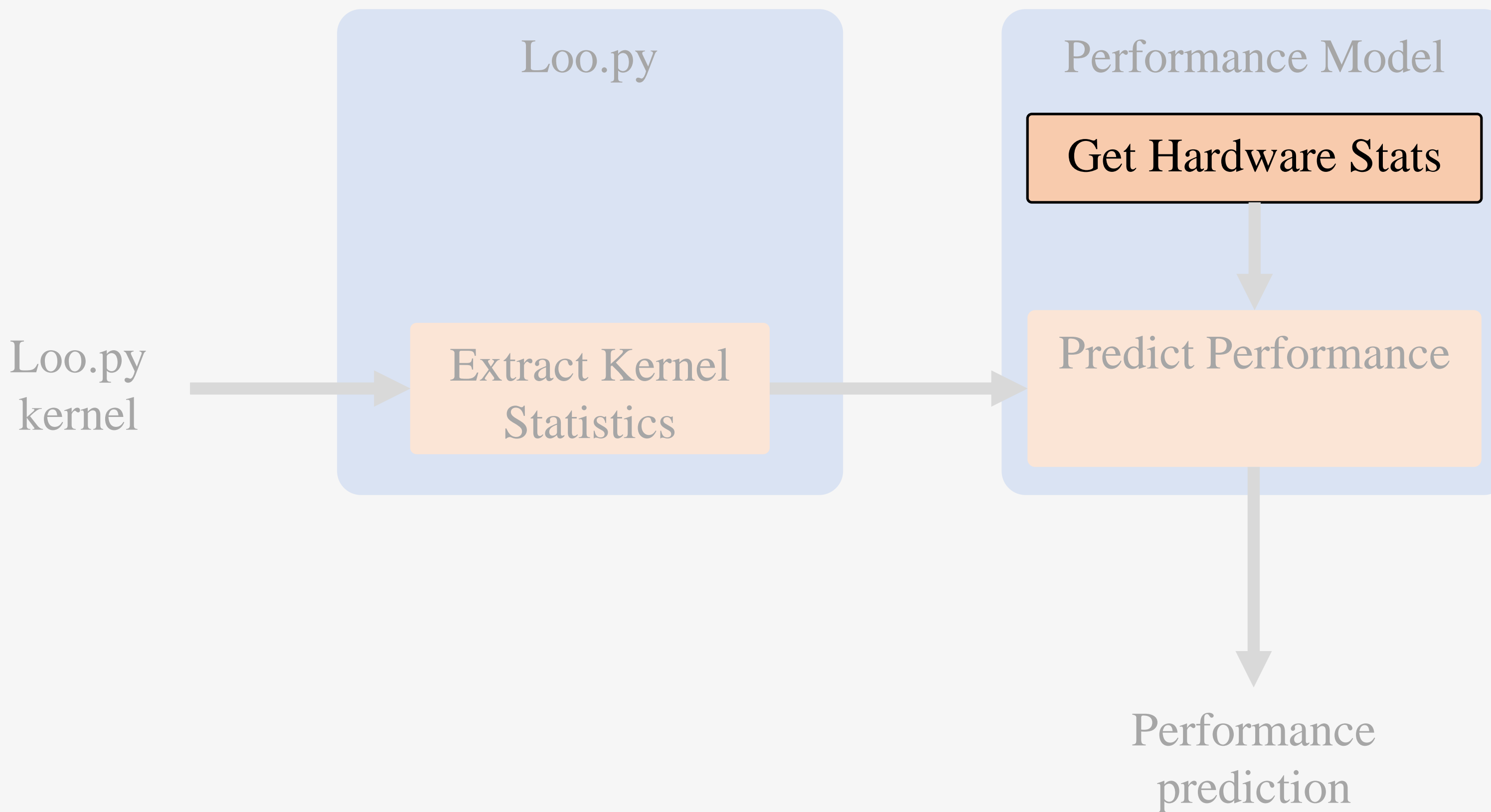
} TBD

- Roadmap

Integration with Loo.py



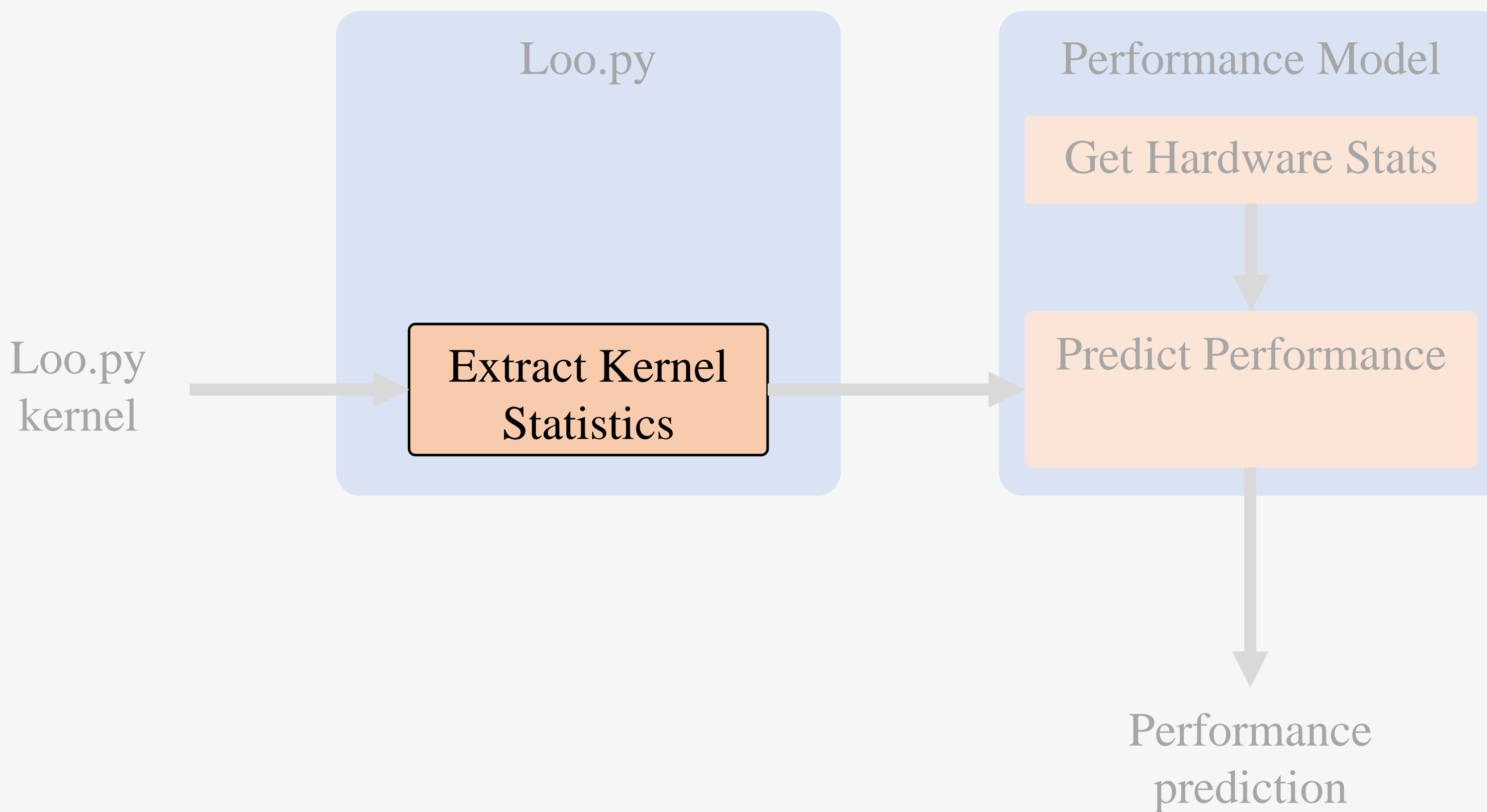
Integration with Loo.py



Gathering Hardware Statistics

- Look up system hardware in static collection of GPU/CPU info
  - Not scalable
- Run empirical tests to gather hardware data before first use on every system
  - Possible, but time consuming
  - Redundant if hardware has already been tested
- Maintain repository of hardware statistics, check for existing system hardware data, use existing data if present, otherwise run tests and update repository

Integration with Loo.py



Extracting Kernel Statistics

- Loo.py kernels contain
  - *Arguments*
  - *Domains*
  - *Iname Implementation Tags*
  - *Instructions*
    - Assignments, declarations, conditionals
- *Instructions* contain *expressions*
  - Slight superset of expressions supported by Pymbolic (symbolic manipulation library built by Dr. Klöckner)
  - Arithmetic expressions, polynomials, reductions

Extracting Kernel Statistics

- Examine instructions and automatically extract values needed for performance model
  - FLOP count
  - Global (DRAM) memory accesses
    - Coalesced and uncoalesced
  - Shared memory usage
  - Register usage
  - Conditional count

- Produce **symbolic output** in terms of problem size
  - Performance model will also produce symbolic output

Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

} TBD

- Roadmap

Outline

- Background and motivation
- Performance model
- Gathering model input
- Results
- Conclusions
- Future Work

} TBD

- Roadmap

Roadmap

- In progress
  - Implement Loo.py kernel statistics gatherer
  - Implement Hong-Kim model, run tests
- Next
  - Incorporate additional factors into model
  - Evaluate additional related work
    - 2010, Baghsorkhi *et al.*, UIUC: *An adaptive performance modeling tool for GPU architectures*
      - Considers control flow divergence and memory bank conflicts
- Future
  - Extend model to include multi-core CPUs, Xeon Phi
  - Build repository for hardware info

Hong-Kim Performance Model

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

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

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

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