Black-Box Kernel-Level Performance Modeling for GPUs

James Stevens <jdsteve2@illinois.edu> Andreas Klöckner <andreask@illinois.edu>
Department of Computer Science · University of Illinois at Urbana-Champaign · Urbana, IL

Abstract

We present a mechanism to **automatically** gather symbolic performance-relevant operation counts from GPU kernels expressed in the Loopy programming system, and apply these counts in a **simple**, **linear model** of kernel run time.

- We use a series of 'performance-instructive' kernels to **fit the parameters** of a unified model to the performance characteristics of GPU hardware from **multiple hardware generations and vendors**.
- We evaluate the predictive power of the model on an array of computational kernels relevant to scientific computing.
- Our simple, **vendor- and GPU-type-independent** model achieves relative accuracy comparable to that of previously published work using *hardware-specific* models.

Modeling Execution Time

Model execution time as linear combination of kernel properties

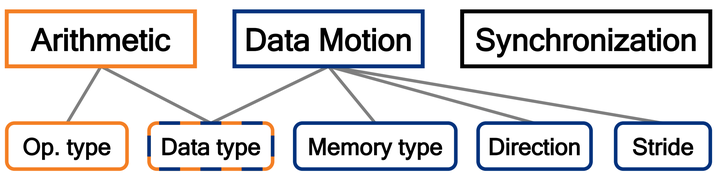
$$T_{\text{wall}}(\mathbf{n}) \approx \sum_{i=1}^{N_{\text{properties}}} \alpha_i p_i(\mathbf{n}),$$

where $\mathbf{n}$ is a parameter set governing problem size and α_i is the weight (run time cost) for the i^{th} property.

Properties

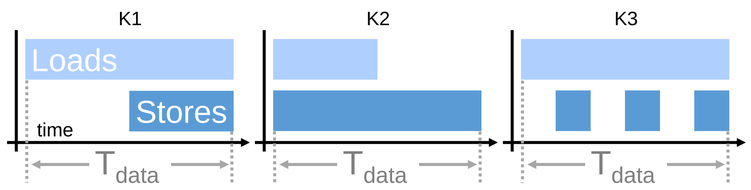
What contributes linearly to kernel execution time?

Three execution cost categories



Five cost factors used to sort kernel stats into properties

Can a **linear** model account for **nonlinearities**? Yes!
Example: On a GPU, global loads and stores may overlap.



Runtime is nonlinear in load/store count. We can model this nonlinear relationship with a combination of three properties:

Want: $T_{\text{data}} = t_{\text{loads}} + t_{\text{stores}} - \min(t_{\text{loads}}, t_{\text{stores}})$
Model: $T_{\text{data}} \approx \alpha_l n_{\text{loads}} + \alpha_s n_{\text{stores}} + \alpha_m \min(\mathbf{n}_{\text{loads}}, \mathbf{n}_{\text{stores}})$

We expect weights $0 < \alpha_l \approx \alpha_s \approx -\alpha_m$

Measurement Kernels

Property costs are revealed through execution of carefully chosen measurement kernels:

- Vector addition (add four vectors)
- Vector copy; store; stride-{1, 2} scale and add
- Filled stride-{2, 3} vector sum reduction (stride-{2, 3} access, but use all data)
- Non-square {tiled, naive} matrix multiplication
- Transpose (with and without prefetching)
- One arithmetic kernel per arithmetic property
- Empty kernel

Property Matrix

One column per **property** / Property **weights**

One row of properties per **measurement kernel**

$$\begin{bmatrix} \frac{p_1^1(n_1)}{T_1} & \cdots & \frac{p_{N_{props}}^1(n_1)}{T_1} \\ \vdots & \ddots & \vdots \\ \frac{p_1^{N_{knls}}(n_{N_{knls}})}{T_{N_{knls}}} & \cdots & \frac{p_{N_{props}}^{N_{knls}}(n_{N_{knls}})}{T_{N_{knls}}} \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_{N_{props}} \end{bmatrix} \approx \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

Divide properties by measurement kernel run times so linear least squares finds weights **minimizing relative error**.

Loo.py: Transformation-based code generation for GPUs and CPUs

Loo.py, a programming system embedded in Python, meets the challenge of **heterogeneous computing** by defining a **data model** for array-style computations and a library of **transformations** that operate on this model.

loo.py
what would you like to unroll today?

Transformations: Loop tiling, instruction-level parallelism, vectorization, unrolling, prefetching, AoS↔SoA, and more!

Specify **mathematical intent**:

```
knl = make_kernel(
 "{ [i,j]: 0<=i<n and 0<=j<m }", # domain
 "out[i,j] = 2*a[i,j]+b[i,j]", # instr. 1
 assumptions="n,m >=1")
```

Specify **transformations**:

```
knl = split_iname(knl, "i", 128,
       outer_tag="g.1", inner_tag="l.1")
knl = split_iname(knl, "j", 128,
       outer_tag="g.0", inner_tag="l.0")
```

Gathering Kernel Statistics

1. Recursively traverse **instruction expression tree** of a Loopy kernel, counting stats for single instruction
2. Determine how many times instruction executes

Instruction 1 (above) contains

- 2 32-bit stride-1 float loads
- 1 32-bit stride-1 float store
- 1 32-bit float multiplication
- 1 32-bit float addition

and executes $n * m$ times.

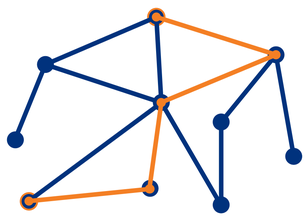
Statistics dict:

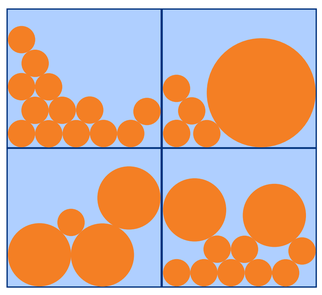
```
{'f32s1L': 2*n*m,
 'f32s1S': n*m,
 'f32-mul': n*m,
 'f32-sum': n*m}
```

Applications

In **performance optimization**, aid in **exploring search space** of program transformations.

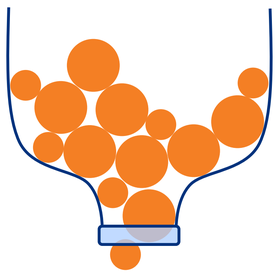
In **algorithm design**, identify **largest contributors to computational cost**.





In **load balancing**, accurate predictions of workload run times enable **better scheduling decisions**.

In **machine bringup and qualification**, our measurement procedure can **expose bottlenecks** and unexpected interactions, and help compare processor architectures.



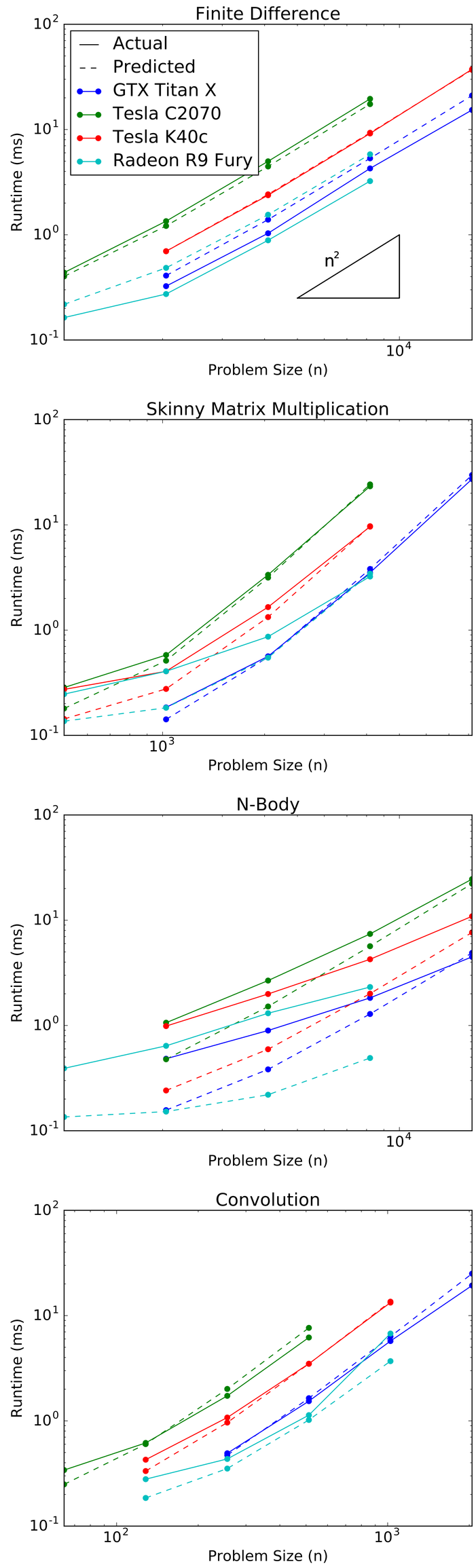
Contributions

- A set of **hardware-independent kernel properties** that account for kernel run times with considerable accuracy.
- A procedure for **automatic extraction of kernel statistics** as piecewise quasi-polynomials.
- A set of **measurements** and a **fitting procedure** to, in a black box and unassisted fashion, determine hardware-specific weights for each property.

References

[1] Klöckner, Andreas. *"Loo.Py: Transformation-based Code Generation for GPUs and CPUs"* Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming, ARRAY'14, 82:82-82:87.

Results



Actual vs. predicted execution times (ms) for four test kernels on four GPUs

Accuracy

Kernel	Nvidia GTX Titan X	Nvidia Tesla C2070	Nvidia Tesla K40c	AMD Radeon R9 Fury	Cross-GPU Geo-Mean
Finite Diff	0.30	0.10	0.01	0.63	**0.11**
Skinny MM	0.08	0.10	0.13	0.28	**0.13**
N-Body	0.32	0.27	0.54	0.76	**0.43**
Convolution	0.10	0.13	0.03	0.23	**0.10**
Cross-Kernel Geo-Mean	**0.16**	**0.14**	**0.06**	**0.42**	

Geometric means of relative error in model prediction