A Unified, Hardware-Fitted, Cross-GPU Performance Model

James Stevens
Andreas Klöckner

May 2, 2016

Goal

- Predict performance of computational kernels on GPUs

Related Work

- Requires detailed hardware knowledge
- Requires instruction-level analysis of code
  - Often by hand
- Demonstrated on single GPU or GPUs of same vendor and generation
- Achieves wide range of accuracy, generally no better than about 12% geometric mean error

Goal

- Predict performance of computational kernels on GPUs

Goal

- Predict performance of computational kernels on GPUs
  - Without hardware knowledge
  - Across hardware vendors/generations
  - Automatically
  - Quickly
  - Simply

- How much accuracy must be sacrificed?

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 parameter set governing problem size and α_i is weight (run time cost) for i^{th} property

Outline for Model Discussion

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

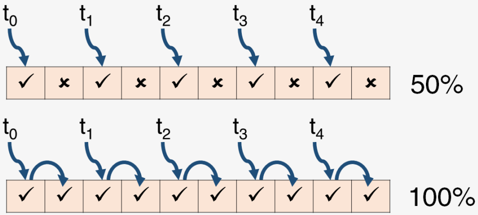
1. Which properties p_i contribute linearly to execution time?
2. How do we gather kernel statistics to produce properties?
3. How do we determine hardware-specific property weights α_i?

Modeling Execution Time

- Kernel Property Categories
  - Data motion
  - Arithmetic
  - Synchronization
  - Launch overhead

Data Motion Properties

1. Global memory access counts
   - Categorize by stride, data type, direction
   - Include min(*loads*, *stores*) property to account for nonlinearity from overlapping loads and stores
   - Further categorize strided access by array utilization percentage



2. Local (shared) memory access counts
   - Categorize by data type, direction

Arithmetic Properties

1. Arithmetic operation counts
   - $+/-$
   - $*$
   - $\div$
   - $\wedge$ (separate property for small integer powers like a^2)
   - Special operations, e.g., rsqrt()

   - Categorize by data type

Synchronization Properties

1. Barrier counts
   - Total encountered by all threads

Launch Overhead Properties

1. Constant (i.e. $p_{const}(\mathbf{n}) = 1$) for kernel launch overhead
2. Thread group count for additional group launch overhead

How do we gather these statistics automatically?

Loopy

- Programming system embedded in Python that enables creation of transformable computational kernels for GPUs
- Motivation: separate mathematical intent from computational minutiae

Loopy

- Example: matrix multiplication

Specify mathematical intent:

```
kn= make_kernel(
      "{[i,k,j]: 0<=i<n and 0<=k<m and 0<=j<l}", # loop domain
      "c[i, j] = sum(k, a[i, k]*b[k, j])" # instructions
      , name="matmul", assumptions="n,m,l >= 1")
```

Specify transformations:

```
# parallelize i and j loops
kn= split_iname(kn, "i", 16, outer_tag="g.0", inner_tag="l.1")
kn= split_iname(kn, "j", 16, outer_tag="g.1", inner_tag="l.0")
```

Extend Loopy - Kernel Stats Counting

- Examine Loopy's internal representation of kernel
- To count memory accesses
  1. For each instruction,
     1.1 Recursively traverse expression tree, accumulating mem. accesses in mapping of category tuples to counts, e.g., $\{(dtype, stride, direction, arrayname) : count\}$
     1.2 Determine number of repetitions in terms of kernel parameters ($\mathbf{n}$) by examining loop index domains
     1.3 Multiply counts in mapping by polynomial of kernel parameters
  2. Accumulate total for all instructions
- Similar process for counting arithmetic operations

Extend Loopy - Kernel Stats Counting

- To count barriers
  1. Generate ‘scheduled’ Loopy kernel
     - Determines ordering/nesting of loops
  2. Step through instructions counting barriers, keeping track of repetition incurred when entering loops
  3. Again, return polynomial in terms of parameters $\mathbf{n}$

Fitting Model

- We now have $p_i(\mathbf{n})$ for all i

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

- How do we find weights α_i?

Fitting Model

- Run set of cleverly designed measurement kernels
- Collect execution times for each kernel, store properties in matrix A with one property per column
- Use LLS to find weights α_i minimizing *relative* error

Minimizing Relative Error

One column per property

One row per kernel

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

Minimize *relative* error

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

Measurement Kernel Set

1. Non-square matrix multiplication (tiled and naive)
2. Transpose (with and without prefetching)
3. Vector scale and add (stride-1 and stride-2 access)
4. Perform arithmetic (one kernel for each arithmetic property)
5. Vector copy
6. Vector addition (add four vectors)
7. Vector store (no loads, just store index in each element)
8. Filled stride-2 vector sum reduction (stride-2 access, but use all data)
9. Filled stride-3 vector sum reduction (stride-3 access, but use all data)
10. Empty kernel

Measurement Kernel Set

- For each kernel configuration,
  - 4 to 8 problem sizes
  - 3 thread group configurations
- 360 measurement kernels total

Test Kernels

1. Finite Differences
   - Applies 5-pt stencil w/ quadratic source term to square matrix
   - Prefetches $(gsize + 2) \times (gsize + 2)$ tiles into local mem.
2. 'Skinny' Matrix Multiplication
   - Performs tiled multiplication of two matrices of size $n \times m$ and $m \times l$, where $n = l = m/8$
   - Prefetches $gsize \times gsize$ tiles into local mem.

$$[\quad A \quad]\begin{bmatrix} \\ B \\ \\ \end{bmatrix} = [C]$$

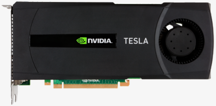
Test Kernels

3. Convolution
   - Applies three 7×7 image filters to three square RGB images
   - Prefetches $(gsize + 6) \times (gsize + 6)$ image tiles into local mem.
   - Stores filters in local mem.

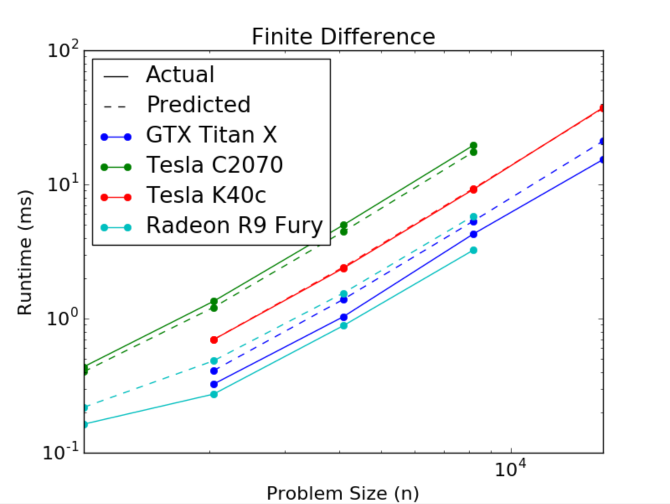
4. N-Body
   - Given $3 \times n$ array of n positions (column-major data layout), computes sum of inverses of distances between each position and every other position
   - Prefetches $3 \times gsize$ tiles into local mem.

Test Hardware

- GPUs
  1. Nvidia GTX Titan X (Maxwell generation)
  2. Nvidia Tesla K40c (Kepler generation)
  3. Nvidia Tesla C2070 (Fermi generation)
  4. AMD Radeon R9 Fury

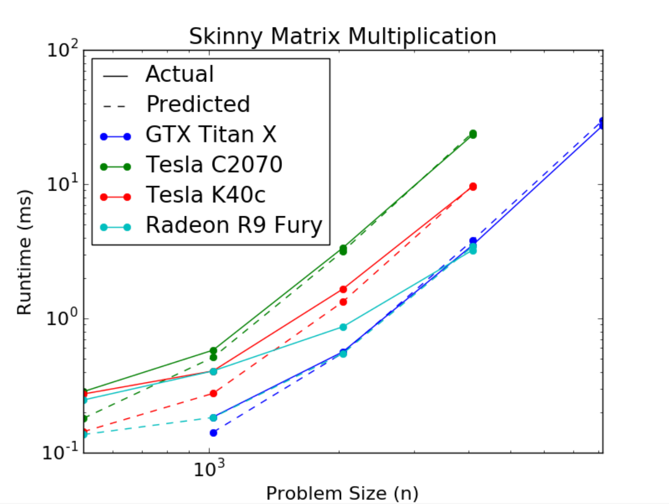


Results: Finite Differences



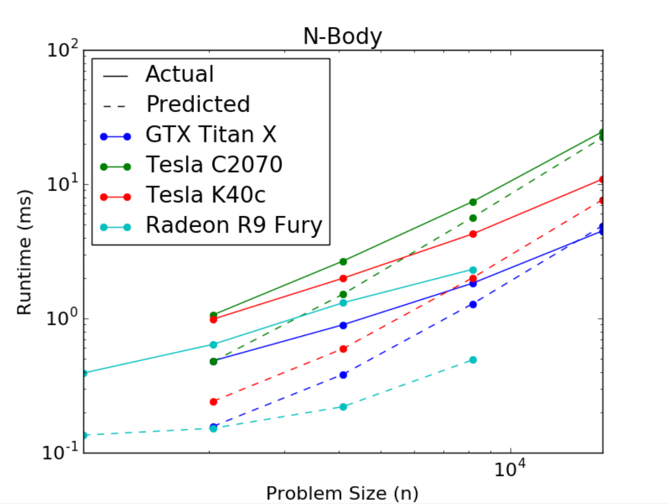
Geo-mean	Error
Titan X	0.30
C2070	0.10
K40c	0.01
R9 Fury	0.63
Overall	**0.11**

Results: Skinny Matrix Multiplication



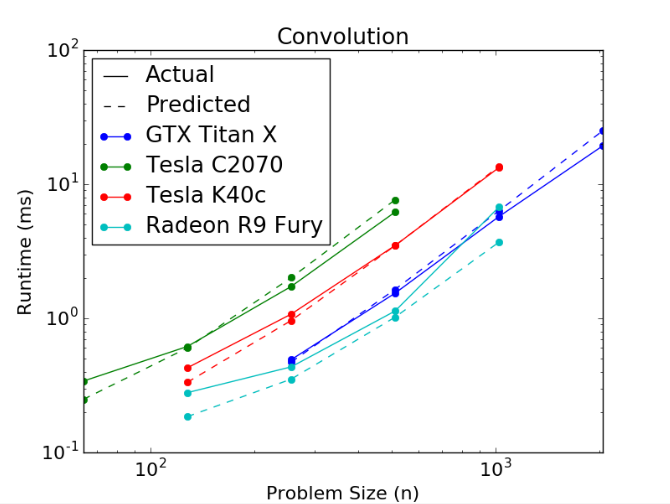
Geo-mean	Error
Titan X	0.08
C2070	0.10
K40c	0.13
R9 Fury	0.28
Overall	**0.13**

Results: N-Body



Geo-mean	Error
Titan X	0.32
C2070	0.27
K40c	0.54
R9 Fury	0.76
Overall	**0.43**

Results: Convolution



Geo-mean	Error
Titan X	0.10
C2070	0.13
K40c	0.03
R9 Fury	0.23
Overall	**0.10**

Accuracy Summary: Geo-Means of Rel. Abs. Error

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**	

Example Weights - Radeon R9 Fury

Property	Weight
Addition/Subtraction	6.81e-13
Multiplication	5.68e-13
Exponentiation (only squaring)	3.91e-13
Other Ops (only rsqrt)	1.61e-12
Local F32 Loads	-1.76e-12
F32 Stride-1 Loads	8.27e-12
F32 Stride-2 (100%) Loads	9.82e-13
F32 Stride-3 (33%) Loads	2.89e-11
F32 Stride-3 (100%) Loads	9.30e-13
F32 Uncoalesced (100%) Loads	2.67e-12
F32 Stride-1 Stores	6.52e-12
F32 Uncoalesced (100%) Stores	3.55e-10
Min(Stride-1 Loads, Stride-1 Stores)	-6.63e-12
Barriers	4.26e-11
Thread Groups	3.75e-09
Const(1)	1.29e-04

Comparison to Related Work

- Differences:
  - We completely automate gathering of all performance-relevant kernel properties used in model
  - We model execution time without explicit representation of hardware characteristics or behavior
  - Our model is hardware vendor- and generation- independent
  - Our model is simple and amenable to analysis; weights have known meanings, allowing reasoning about sources of kernel execution cost
  - Our model evaluation is rapid and simple, requiring small inner-product

Potential Applications

- *Performance Optimization*
  Selecting fastest kernel in kernel configuration space
  - Runtime performance tuning
- *Algorithm Design*
  Providing programmer with insight into which aspects of workload contribute most to cost
- *Load Balancing*
  Providing accurate predictions of workload run times enabling better scheduling

End

Questions?