Accelerating cuBLAS/cuDNN using Input-Aware Auto-Tuning

The ISAAC library

Philippe Tillet
Harvard University

Introduction

cuBLAS does not always achieve peak performance:

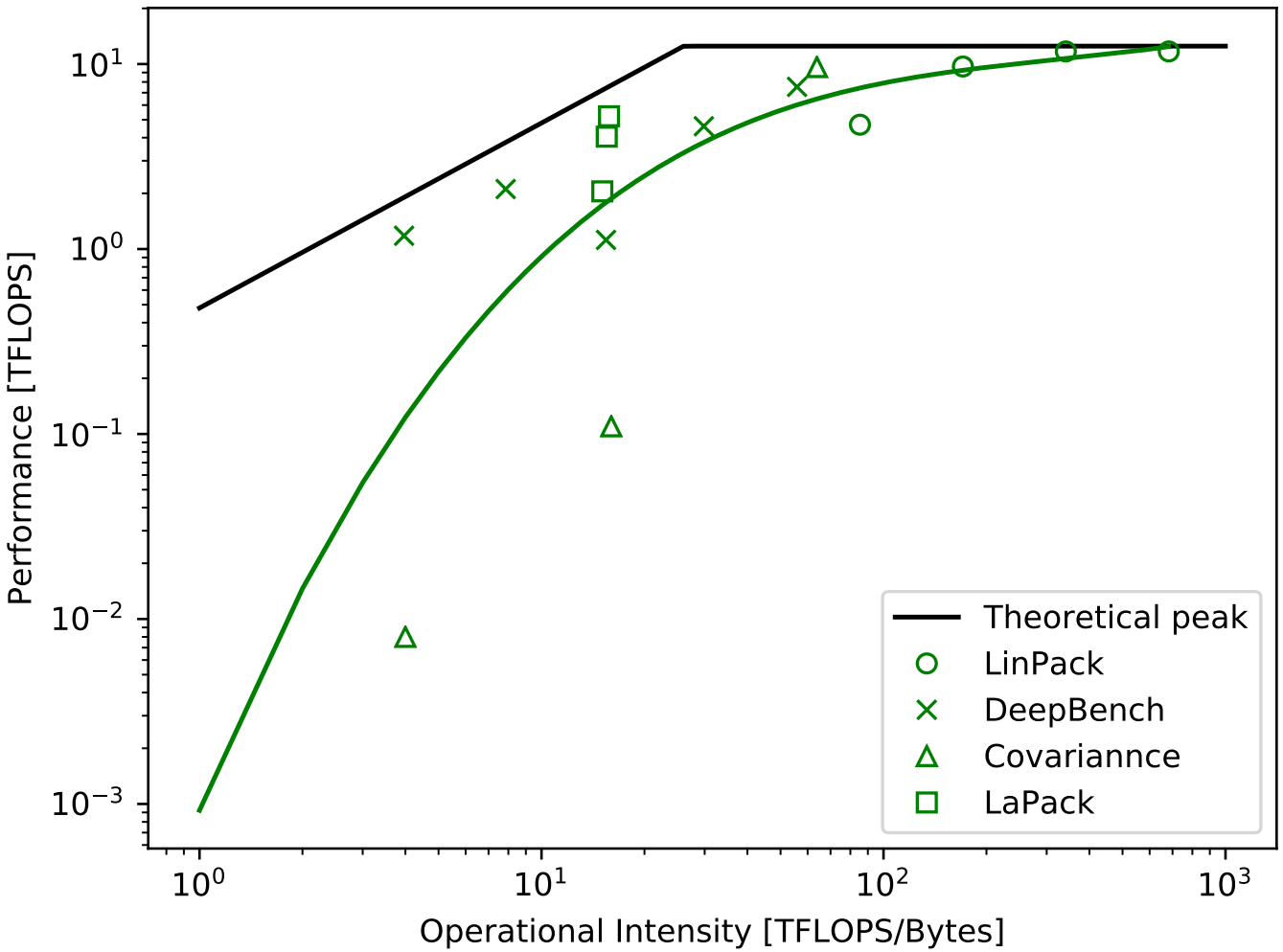
- (M, N, K)[1] = (4096, 4096, 4096): 95%
- (M, N, K) = (1760, 32, 1760): 15%
- (M, N, K) = (16, 16, 128,000): 0.1%

Yes, some configurations are IO-bound, but still...

[1] Product of an $M \times K$ by a $K \times N$ matrix

Introduction

Figure: cuBLAS (GEMM) vs Roofline Model – Pascal Titan X



Introduction

cuBLAS/cuDNN are good...

- Better than anything else so far
- Achieves peak performance ... sometimes

... but not perfect

- Lack performance-portability (across hardware/tensor shapes)

Can we do better?

Method

Performance portability across hardware is a solved problem:
Assume the existence of a kernel generator for GEMM/CONV
$\mathbf{x}_k$: kernel parameters (e.g., tile sizes)
$\mathbf{x}_i$: input parameters (e.g., tensor shape, data-type)
$y(\mathbf{x}_i, \mathbf{x}_k)$: Performance of a given kernel on given inputs

Auto-Tuning (ATLAS, clBLAS, etc.):

- Offline: Choose $\mathbf{x}_i$; find $\arg\max_{\mathbf{x}_k} y(\mathbf{x}_i, \mathbf{x}_k)$.

Method

ISAAC adds input portability:

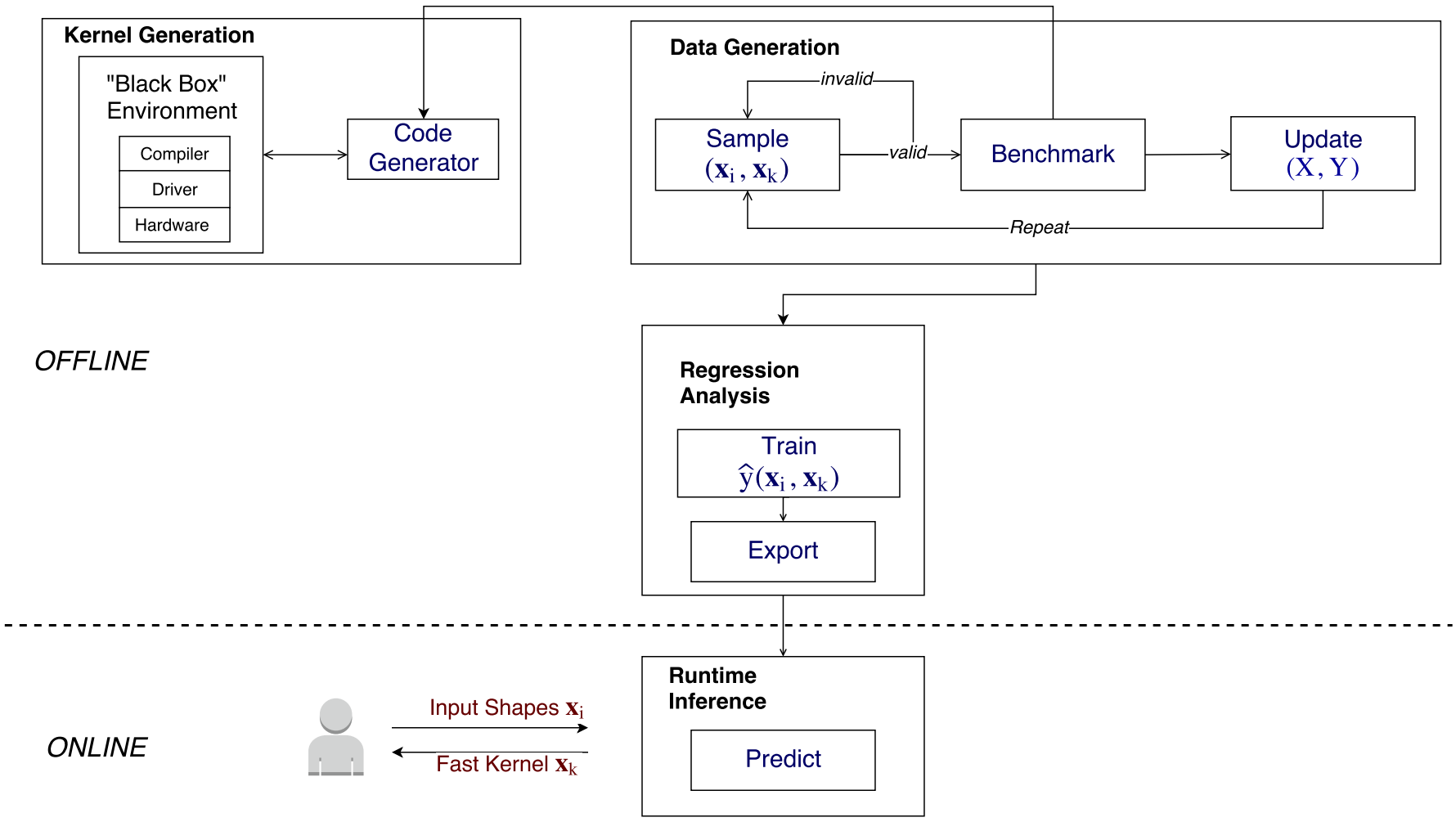
- We want to retain good performance across the entire space of inputs

Input-Aware Auto-Tuning:

- Offline: Build a predictive model $\hat{y}$ for y.
- Online: $\mathbf{x}_i$ is imposed; find $\arg\max_{\mathbf{x}_k} \hat{y}(\mathbf{x}_i, \mathbf{x}_k)$.

Method

Figure: Flowchart of ISAAC



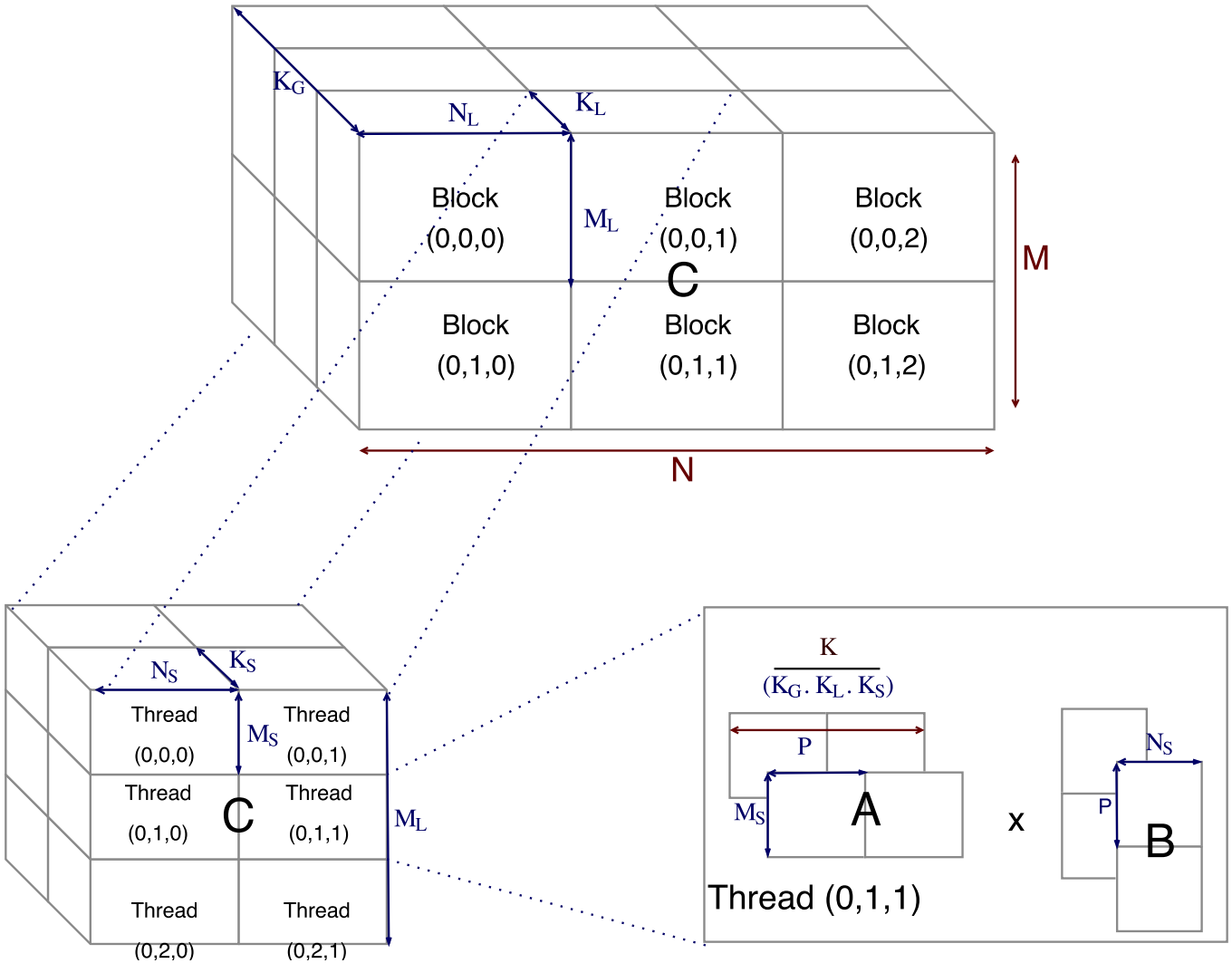
Kernel Generation

Goal: Transform kernel parameters $\mathbf{x}_k$ into functional binaries.

Typical kernel parameters are tile sizes, reduction splits, pre-fetching factors.

Kernel Generation

Figure: Parameterization of GEMM
$\mathbf{x}_k = (M_L, N_L, M_S, N_S, P, K_G, K_L, K_S)$



Kernel Generation

Implementation details

- Double-buffered memory loads
- Vector loads/stores are used when possible
- CONV is essentially GEMM with a look-up table
- PTX code generation:
  - Faster compilation i.e., auto-tuning
  - No CUDA SDK dependency
  - $\sim 20 - 30\%$ performance gain vs CUDA C (predicates)

Data Generation

Goal: Generate a set of pairs $(\mathbf{x}_n, y_n)$ where $\mathbf{x} = (\mathbf{x}_i, \mathbf{x}_k)$

Method: Sample $\mathbf{x}$ and measure y.

- About 99.9% of the generated configurations are invalid!
- Build a generative model for valid $\mathbf{x}$

Regression Analysis

Goal: Given X, Y build a predictive model $\hat{y}(\mathbf{x})$

Method:

- MLPs are a good choice because:
  - Generating data-points is cheap
  - Fast, batched inference

- Vanilla ML algorithms are not good at handling multiplications/divisions
  $\Rightarrow$ Feature transormation $\mathbf{x}' = \log \mathbf{x}$

Runtime Inference

Goal: Given $\mathbf{x}_i$, find the best possible $\mathbf{x}_k$.

Method: Compute $\arg\max_{\mathbf{x}_k} \hat{y}(\mathbf{x}_i, \mathbf{x}_k)$.

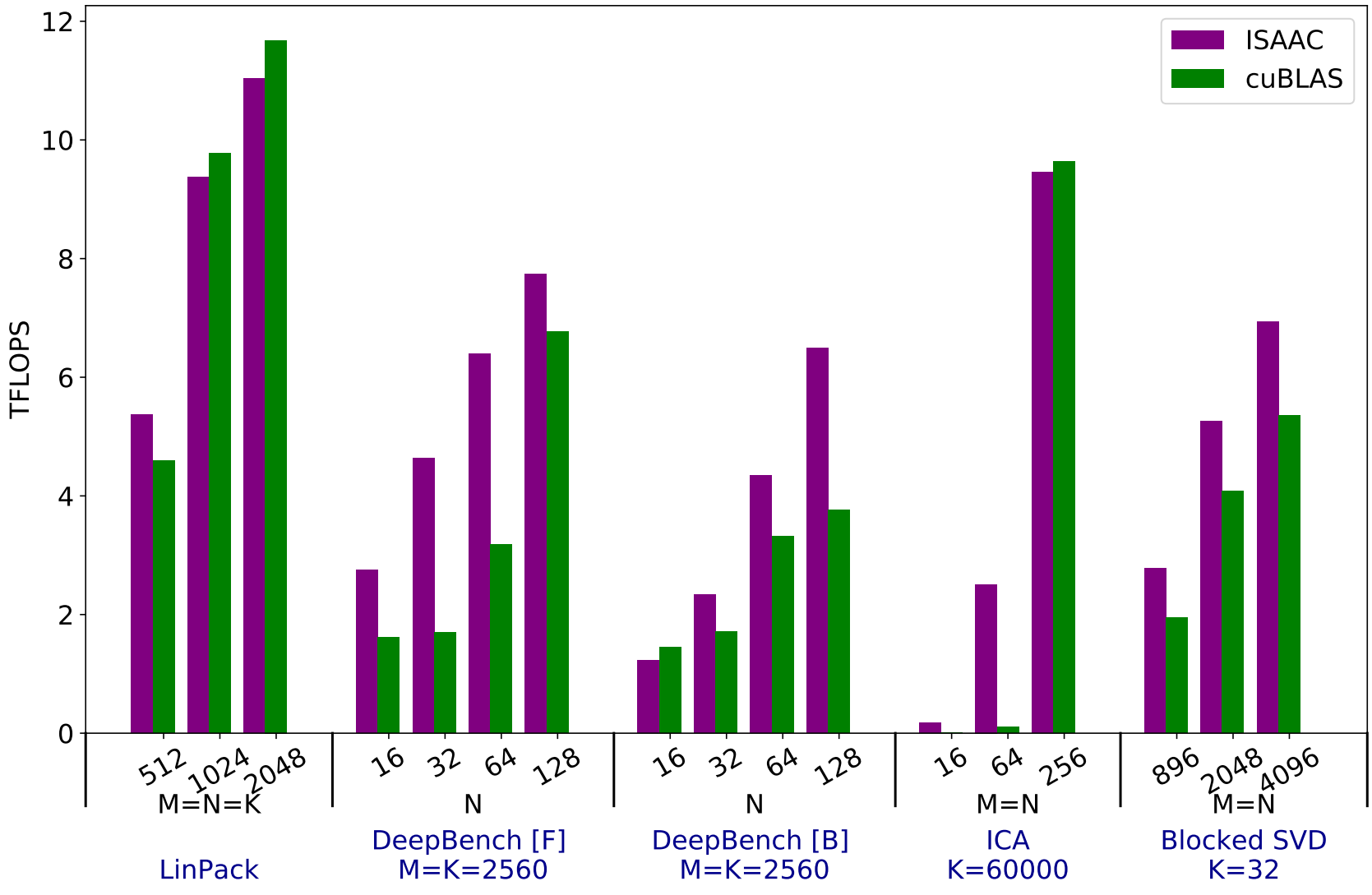
- Exhaustive search: millions of candidates $\mathbf{x}_k$ can be evaluated in one second. Global maximum guaranteed
- Other choices: GA, Simulated Annealing...
- Re-benchmark the ~ 10 best predictions and pick the actual fastest.

Method Summary

- Build a parameterized code generator for GEMM and CONV
- Benchmark random kernels on random input configurations
- Build a predictive model for the performance of any kernel on any shape
- For a fixed shape, maximize the model over kernels.

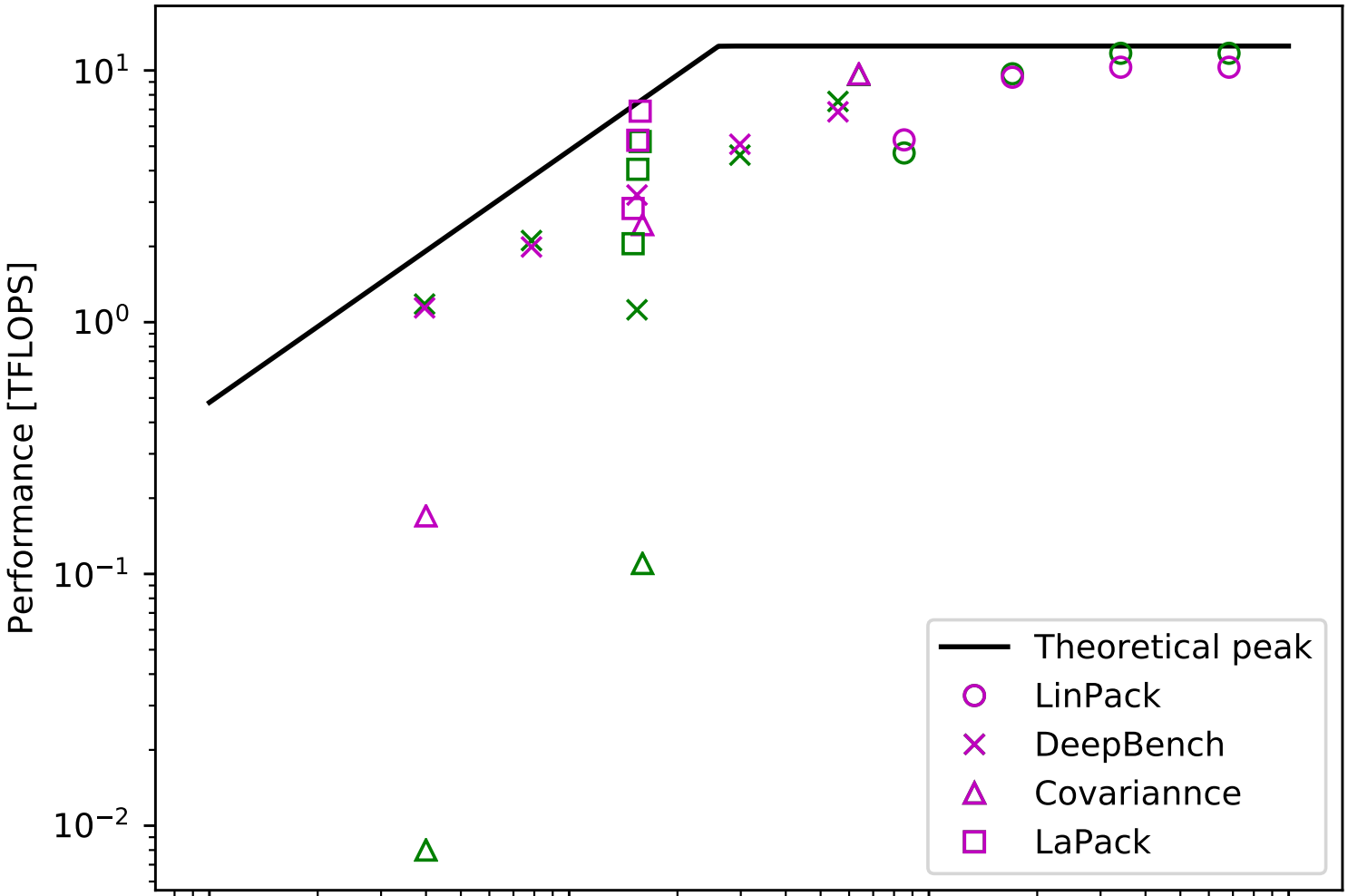
Benchmarks

Figure: SGEMM on TitanX (Pascal)



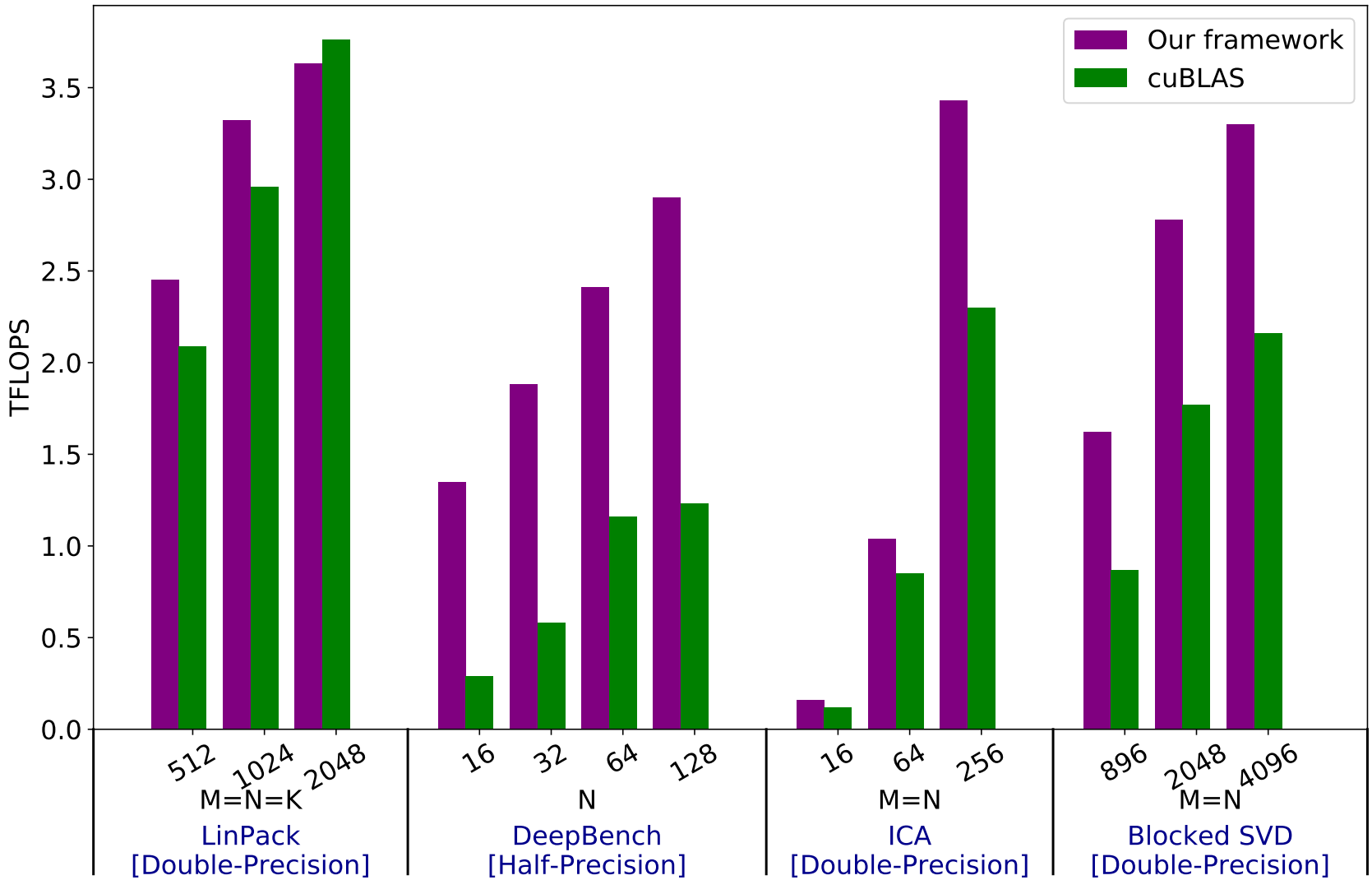
Benchmarks

Figure: Roofline Model - Revisited



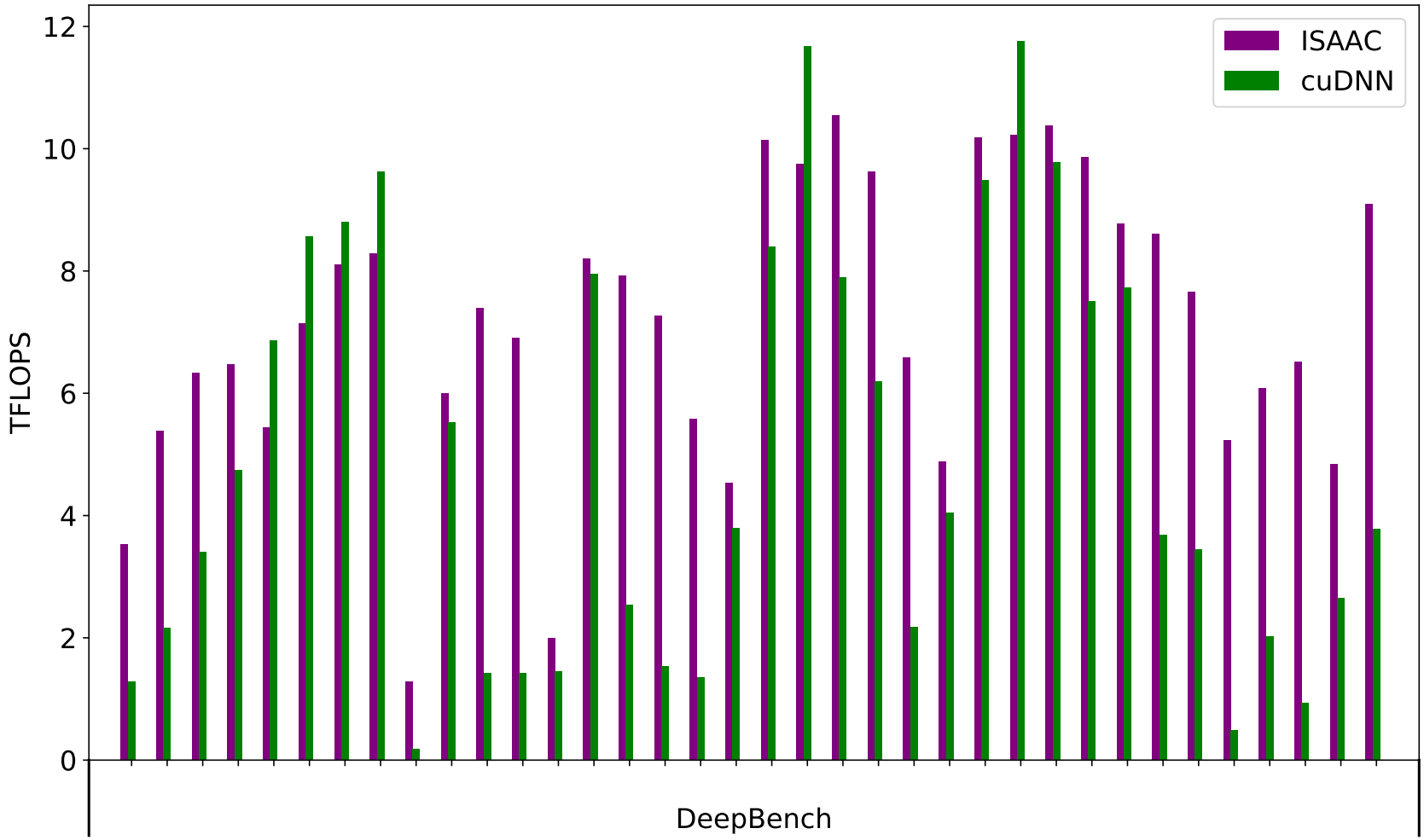
Benchmarks

Figure: HGEMM/DGEMM on P100



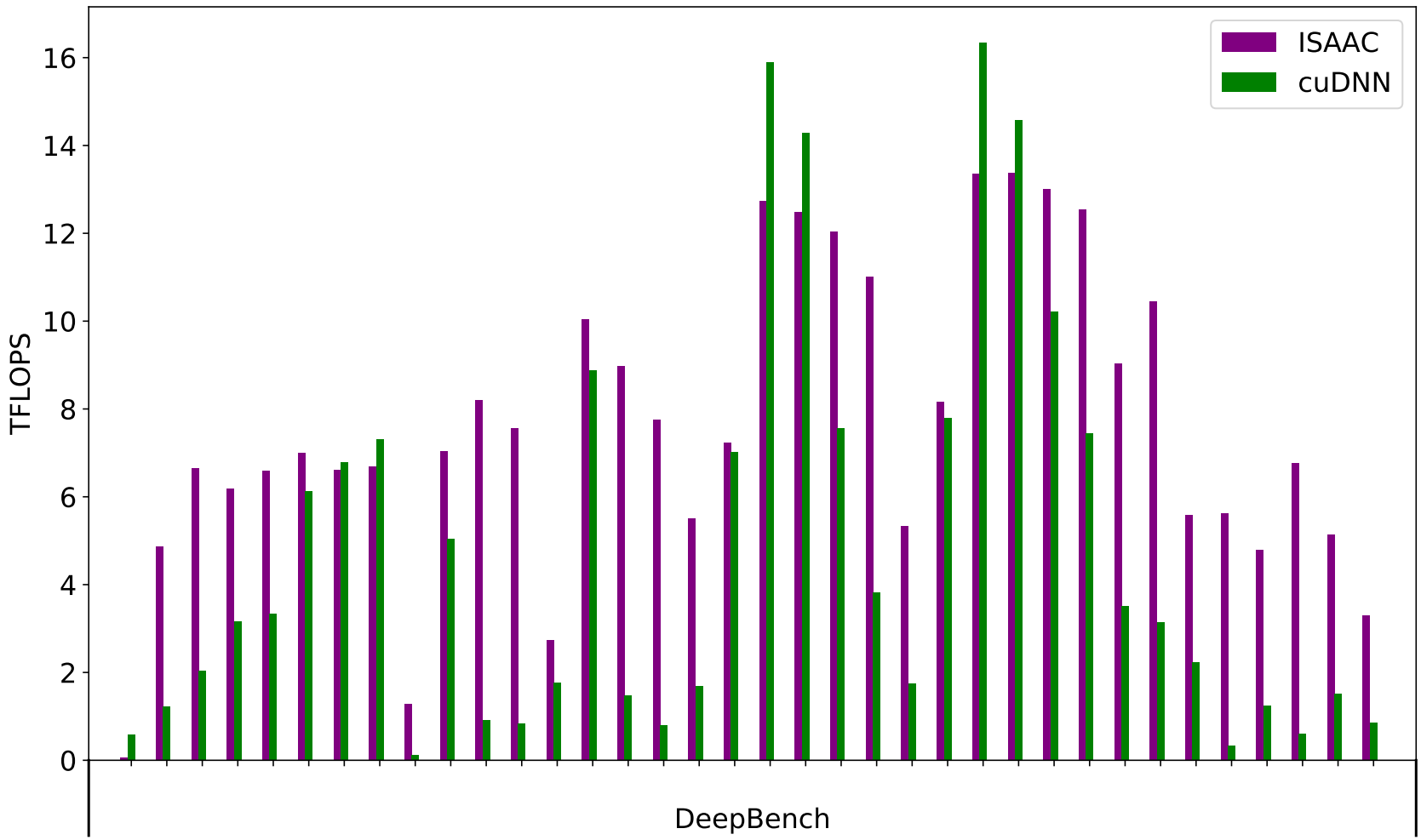
Benchmarks

Figure: SCONV on TitanX (Pascal)



Benchmarks

Figure: HCONV on P100



Conclusions

- Presented the design and implementation of ISAAC
- Performance improvements of 0.8 - 9x over cuDNN
- Performance improvements of 0.9 - 3x ($>$ 30x on ICA) over cuBLAS
- Fast release cycle (auto-tuning takes $\sim$ 3 hours)
- git clone -b v2.0 `https://github.com/ptillet/isaac.git`

Thanks for your attention!

Benchmarks

Figure: SGEMM on GTX980