Improving Polyhedral-Based Optimizations With Dynamic Coordinate Descent

Gaurav Verma*, Michael Canesche†, Barbara Chapman*, Fernando Magno Quintão Pereira†
*Stony Brook University, USA, {gaurav.verma, barbara.chapman}@stonybrook.edu
†UFMG, Brazil, {michaelcanesche, fernando}@dcc.ufmg.br

***Abstract*—Polyhedral optimizations have been a cornerstone of kernel optimization for many years. These techniques use a geometric model of loop iterations to enable transformations like tiling, fusion, and fission. The elegance of this approach lies in its ability to produce highly efficient code through fully static optimizations. However, modern kernel schedulers typically avoid the polyhedral model, opting instead for dynamic sampling techniques, such as evolutionary searches, to generate efficient code. The polyhedral model is often bypassed because, being entirely static, it struggles to adapt to the fine details of hardware. In this work, we demonstrate that it is possible to overcome this limitation by combining the polyhedral model with a post-optimization phase based on dynamic coordinate descent, which uses minimal sampling while still achieving excellent performance.**

***Index Terms*—polyhedral model, coordinate descent, kernel optimization, search space**

I. Introduction

The polyhedral model, introduced nearly 40 years ago [1]–[7], is a mathematical framework for analyzing and transforming loops controlled by affine induction variables. Optimizations within the polyhedral model involve applying transformations that reorder, fuse, distribute, or tile loops to exploit parallelism and enhance data locality.

The anticipated golden age of polyhedral compilation was expected to arrive a few years ago with the realization that computations for training and inference in deep learning networks are inherently polyhedral. However, modern compiler frameworks for deep learning models, such as Google's XLA [8], TVM [9], the ONNX Runtime [10], the PyTorch JIT Compiler [11], and NVIDIA's TensorRT [12], do not fully leverage the analyses that are central to polyhedral optimizations. Instead, they rely on a methodology that we refer to as *feedback-guided*: kernels are generated, sampled, and profiled. Profiling data is then used to inform subsequent transformations to generate improved kernel versions. Sampling is preferred over purely static polyhedral analyses because it makes the compiler *hardware-aware*, enabling it to adjust optimizations based on hardware parameters such as cache sizes and the number of available physical threads.

In this paper, we show that combining polyhedral analyses with sampling can outperform both purely polyhedral and sampling-based techniques. As Section II explains, we use polyhedral analyses to construct an initial *kernel optimization space*, which represents a configuration of loops including their relative ordering and tiling. This optimization space is then sampled using coordinate-descent to determine optimal parameters for tiling, threading, and unrolling. We hypothesize that by treating performance as a function of these parameters, we obtain a convex hyper-surface, making coordinate-descent likely to converge to an optimal configuration.

We have implemented our ideas on top of Pluto, a well-known polyhedral optimizer [13]. Section III shows that coordinate descent improves the code produced by Pluto using six different kernels typically found in deep-learning models. Furthermore, our approach might be up to 20x faster than the exhaustive search of optimization parameters.

II. Overview

Figure 1 shows an implementation of the general matrix multiplication (GEMM) kernel, a core component of deep-learning models. The kernel in Figure 1 shows poor spatial locality. For instance, depending on the dimensions of the matrices, every access of `A[i][k]` and `output_naive[i][j]` will incur in a cache miss.

```
for (size_t i = 0; i < A_height; ++i) {
    for (size_t j = 0; j < B_width; ++j) {
        for (size_t k = 0; k < A_width; ++k) {
            output_naive[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Fig. 1: Naïve general matrix multiplication kernel

Figure 2 shows an optimized version of matrix multiplication. The implementation in Figure 2 was produced by Pluto [13], a polyhedral-based source-code optimizer. In this example, Pluto has transformed the kernel in Figure 1 via two polyhedral-based optimizations: loop interchange and tiling. The new implementation of GEMM achieves higher memory reuse because the induction variables `t4` and `t5` have been bounded to vary within cache limits.

The kernel in Figure 2 is not optimal. It is possible to improve upon it on two architectures, ARM A64FX and Intel Xeon Max, by changing the tiling dimensions. Figure 2 hard-codes these dimensions as the constant 32. However, statically determining the correct tiling dimensions is difficult. To explain why, Figure 3 shows how performance varies with different sizes of tiling windows (we vary the tiling windows of the loops controlled by `t5` (X) and `t4` (Y) in Figure 2).

```
int lbv, ubv;
size_t t1, t2, t3, t4, t5, t6;

for (t1 = 0; t1 <= floord(A_height - 1, 32); t1++) {
  for (t2 = 0; t2 <= floord(B_width - 1, 32); t2++)
      {
    for (t3 = 0; t3 <= floord(A_width - 1, 32); t3
        ++) {
      for (t4 = 32 * t1; t4 <= min(A_height - 1, 32
          * t1 + 31); t4++) {
        for (t5 = 32 * t3; t5 <= min(A_width - 1, 32
            * t3 + 31); t5++) {
          lbv = 32 * t2;
          ubv = min(B_width - 1, 32 * t2 + 31);
          for (t6 = lbv; t6 <= ubv; t6++) {
            output_pluto[t4][t6] += A[t4][t5] * B[t5
                ][t6];
          }
        }
      }
    }
  }
}
```

Fig. 2: Pluto-generated general matrix multiplication kernel

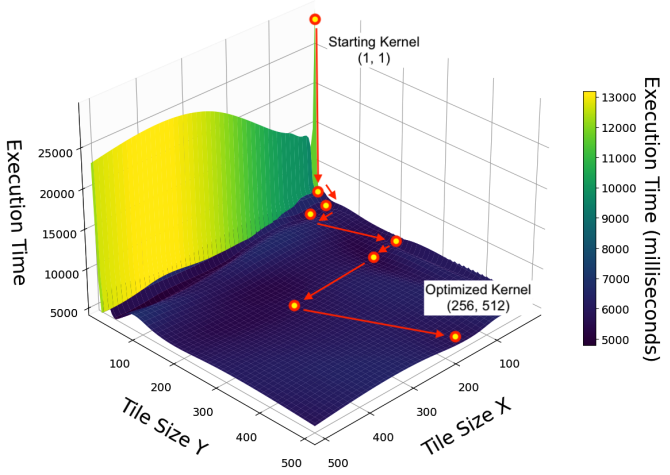

Fig. 3: GEMM Surface Plot.

Finding the optimal parameter configuration is difficult because the surface in Figure 3 has local minima. Yet, the path from the origin of that surface $(X = 1, Y = 1)$ to the optimal configuration $(X = 256, Y = 512)$ traverses exclusively a convex region. This observation is called "*The Droplet Hypothesis*" by Canesche *et al.* [14]. Based on this observation, this paper proposes to add droplet search (coordinate descent) as a final adjustment phase onto kernels produced via polyhedral optimizations as a way to fine-tune optimization parameters. Droplet search is sampling-based: it runs all the kernels that form a *neighborhood* (in the performance surface) of the current best kernel and picks the best candidate. If there is no such best candidate, the search stops.

III. Experimental Evaluation

Even a small number of samples searched with coordinate descent is sufficient to improve on fully static polyhedral optimizations. Figures 4 and 5 show normalized execution times for various kernels on the ARM A64FX and Intel Xeon Max architectures. Pluto optimizations significantly improve the performance of naïve kernels, except in `maxpool` and `softmax`. However, these results are not optimal. Our coordinate-based dynamic search method was able to surpass the optimizations achieved by Pluto in every one of the six kernels that we evaluated.

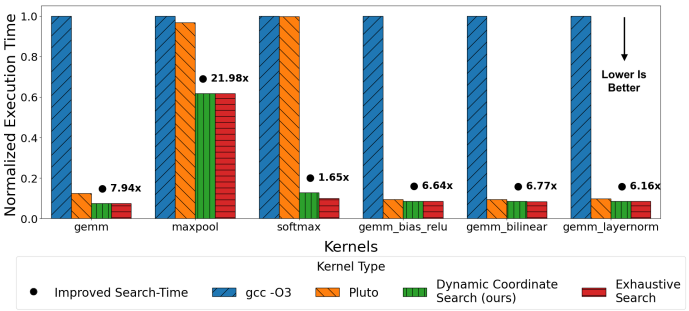

Fig. 4: Comparison of Normalized Execution Times Across Different Kernels on ARM A64FX

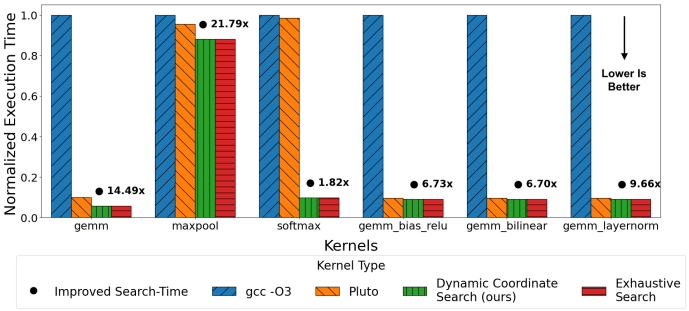

Fig. 5: Comparison of Normalized Execution Times Across Different Kernels on Intel Xeon Max

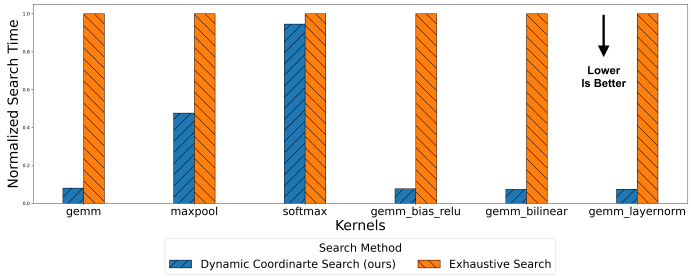

Fig. 6: Comparison of Normalized Search Time Across Different Kernels on ARM A64FX

To emphasize the power of coordinate descent, Figures 4 and 5 also compare it with an exhaustive search. Dynamic

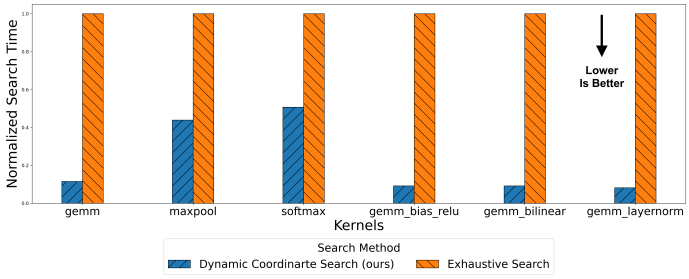

Fig. 7: Comparison of Normalized Search Time Across Different Kernels on Intel Xeon Max

coordinate search converges to similarly optimized kernels. However, the coordinate descent search is much faster. The more optimization parameters available, the larger this difference. Figures 6 and 7 compare these search times. For kernels like GEMM and its variations, which have three or more tiling windows, coordinate descent is more than 10 times faster than exhaustive search.

References

[1] P. Quinton and V. Van Dongen, "The mapping of linear recurrence equations on regular arrays," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 1, no. 2, pp. 95–113, 1989.

[2] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, pp. 23–53, 1991.

[3] ——, "Some efficient solutions to the affine scheduling problem. part ii. multidimensional time," *International journal of parallel programming*, vol. 21, pp. 389–420, 1992.

[4] ——, "Some efficient solutions to the affine scheduling problem, part i," onedimensional time. Technical Report 28, Labaratoire MASI, Institute Blaise, Tech. Rep., 1992.

[5] C. Lengauer, "Loop parallelization in the polytope model," in *International Conference on Concurrency Theory*. Springer, 1993, pp. 398–416.

[6] F. Irigoin and R. Triolet, "Supernode partitioning," in *15th ACM Symposium on Principles of Programming Languages*. ACM, Jan 1988, pp. 319–328.

[7] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, 1993, pp. 126–138.

[8] Google, "Xla overview," https://openxla.org/xla, 2024.

[9] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated {End-to-End} optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 578–594.

[10] Microsoft, "Onnx runtime," https://onnxruntime.ai/, 2024.

[11] J. Ansel, E. Yang, H. He, N. Gimelshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski *et al.*, "Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2024, pp. 929–947.

[12] P. Davoodi, C. Gwon, G. Lai, and T. Morris, "Tensorrt inference with tensorflow," in *GPU Technology Conference*, 2019.

[13] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[14] M. Canesche, V. Rosário, E. Borin, and F. Quintão Pereira, "The droplet search algorithm for kernel scheduling," *ACM Transactions on Architecture and Code Optimization*, vol. 21, no. 2, pp. 1–28, 2024.

Artifact Description Appendix

All the artifacts from this research work are open-source and available on GitHub repository at https://github.com/xintin/kelpie (accessed on 28 September 2024).