Coding Ants – Optimization of GPU Code Using Ant Colony Optimization

Eric Papenhausen and Klaus Mueller

Computer Science Department, Stony Brook University, Stony Brook, NY

Abstract

This article proposes the Coding Ants framework, an approach for autotuning which uses ant colony optimization to find a sequence of code optimizations for GPU architectures. The proposed framework is built as an extension to the PPCG compiler, a source-to-source code generator based on the polyhedral model and specializing in the generation of CUDA code. As such, the Coding Ants framework is able to use the polyhedral abstraction to represent a large space of possible transformations. Several optimizations are also presented which have not been included in any previous GPU autotuning system. The proposed framework also extends the traditional ant colony optimization algorithm to include performance metrics as well as a regression tree analysis to segment the search space. We evaluate the framework on the PolyBench suite and compare the performance of three levels of optimization that transfer increasing control to the Coding Ants framework from the PPCG cost model.

Keywords: Automatic Optimization, GPU Optimization, Autotuning, CUDA, Ant Colony Optimization, Polyhedral Model

1. Introduction

Optimizing performance by hand can be a labor intensive operation. Performance-sensitive applications can require a great deal of effort and experimentation by the programmer. This is especially true for GPU applications, which typically have more architectural features compared to CPUs. Proper utilization of the different memory regions (e.g. global, texture, shared, etc.) and a careful balancing of the thread workload are required to achieve good performance. As a result, subtle changes in the GPU code

can lead to drastic effects on performance. The resulting hand-tuned code often performs well but is optimized for a specific architecture. This makes porting the application to new hardware difficult, as the program needs to be re-tuned for the new architecture.

Iterative optimization, or auto-tuning, has been shown to produce better performance compared to compiler cost models and machine learning methods [27][7]. This typically involves compiling different variants of a program and testing the performance. There is often an intelligent search which uses the previous runs to choose the next program variant. Different algorithms have been used for this task including parallel rank order [51] and best first search algorithms [16].

In this work, we propose the Coding Ants (CA) framework. It is an iterative optimization approach based on the Ant Colony Optimization (ACO) algorithm [9]. Ant Colony Optimization is attractive because it is well suited for graph search problems, and a given computer program code and its variations can be formulated as a graph. Our core idea is to use ACO to optimize computer program code formulated as a graph.

Our CA framework is built off the PPCG (Polyhedral Parallel Code Generator) compiler [57]. It takes C code as input and outputs optimized CUDA code. The CA framework optimizes over a large search space which includes polyhedral transformations (e.g. skewing, permutation, etc.) as well as GPU specific parameters (e.g. thread block size, thread granularity, shared memory usage, etc.). The main contributions of this paper are:

- Demonstrates how the GPU optimization space can be mapped to a, possibly cyclic, graph.
- Presents several optimizations that have not been considered by previous GPU based auto-tuners – including use of parallel reduction and atomic operations.
- Evaluates three optimization levels to determine the optimal work distribution between the PPCG cost model and the CA framework.
- Presents extensions to the ACO algorithm to incorporate performance metrics as a criteria for optimization selection.
- Presents a novel extension to the ACO algorithm which uses a regression tree to segment the search space into regions with promising performance.

– Evaluates the CA framework on the PolyBench suite version 4.2.1 and compares the performance against code generated by the PPCG compiler. Performance is also compared to a random search through the optimization graph.

The remainder of the paper is organized as follows. Section 2 presents background on the GPU architecture and the polyhedral model. Related work is presented in Section 3. Section 4 describes the Polyhedral Parallel Code Generator (PPCG). Section 5 presents the Coding Ants framework along with the optimizations that are considered, and Section 6 discusses extensions to the ACO algorithm. Experiments are presented in Section 7 and Section 8 concludes the paper.

2. Background

2.1. GPU Architecture

The CA framework produces code for NVIDIA GPUs using the CUDA programming language. This is a C like language that allows general purpose GPU (GPGPU) programming. It allows GPUs to be used as an accelerator where the computationally expensive parts of a program can be offloaded and executed in parallel. In CUDA, functions that are executed on the GPU are called CUDA *kernels*.

The GPU architecture is composed of a number of multiprocessors – with each multiprocessor containing a number of processing units, or cores. Each core within a multiprocessor is synchronized and executes in lock-step. As a result, parallelism is exploited in a single instruction multiple thread (SIMT) fashion. Each thread executes the same instruction over different data. Up to three dimensions of parallelism can be exploited. Threads are organized into thread blocks. Thread blocks are further organized into a grid. Threads within a thread block have access to *shared* memory. Shared memory is a type of user controlled cache. Data written to a location in shared memory is visible by all threads in a thread block. A group of 32 threads in a thread block form a *warp*. Warps are scheduled to run on a multiprocessor – with each thread being assigned to a core. In newer GPUs (i.e. starting with the Kepler architecture) intra-warp communication can be performed via *shuffle* instructions. These instructions serve a similar purpose to that of the SSE shuffle instructions. They allow for direct communication between threads in a warp and is a faster method of communication compared to shared memory.

GPUs have several types of memory with different optimal access patterns. Global memory is the largest region and requires that data be accessed in a coalesced fashion. A coalesced access occurs when each thread in a warp accesses adjacent memory locations. A coalesced access pattern results in a single memory request, as opposed to a separate request for each thread. Texture memory is a type of read-only memory in which the texture caches are optimized for 2D spatial locality. This can be particularly useful when the access pattern is unknown at compile time or un-coalesced. Constant memory is also read-only and typically much smaller than texture or global memory. It is designed for small, constant arrays that do not change throughout the execution of a CUDA kernel. Its optimal access pattern is one where every thread in a warp accesses the same memory location.

A key reason that GPUs are so powerful is their ability to perform extremely fast context switches. When a warp stalls (e.g. due to a request to off-chip memory), it can be quickly swapped for a new warp. Context switches are performed quickly because registers are assigned when a thread is created and it does not change throughout the lifetime of the thread. This can, however, cause limitations on parallelism. If each thread requires too many registers, then fewer threads can run concurrently. This results in a lower *occupancy*. Occupancy is a metric for measuring the utilization of multiprocessors and is defined as the ratio of active warps on a multiprocessor to the maximum number of active warps supported by the multiprocessor [2].

2.2. Polyhedral Model

The polyhedral model is a mathematical framework capable of representing static control parts (SCoP) of a program. SCoPs include statements where array accesses, surrounding conditionals and loop bounds are affine functions of outer loop iterators and parameters [8]. Statements are modeled using polyhedra, where affine inequalities are used to represent loop bounds and form a face of the statement's iteration domain. Each point, or iteration vector, in the iteration domain corresponds to an instance in which the statement is executed. For example, Figure 1 shows a loop nest that would give rise to a rectangular model; it would undergo a shearing affine transformation if, say, the start and end indexes of the j-loop were shifted by an integer factor.

Along with the iteration domain, an exact dependence analysis is computed. A dependence occurs when two statement instances access the same memory location, and at least one of those accesses is a write. A generalized

dependence graph (GDG) is computed. The GDG is a directed multi-graph in which nodes correspond to statements and edges correspond to dependences. An edge $e \in E$ in the GDG is represented using a dependence polyhedra, $\mathcal{P}_e$ [21]. Dependence polyhedra models dependences using affine equalities and inequalities and describe the exact statement instances which are dependent.

The polyhedral model is capable of improving performance by scheduling statement instances to execute in such a way that minimizes execution time. This is often achieved by exposing parallelism and improving cache locality. Each statement $s \in V$ in the GDG is scheduled by applying a transformation matrix, H_s. A transformation matrix has d rows with $d + p + 1$ columns, where d is the dimensionality of the statement's iteration domain and p is the number of program parameters. Each row, ϕ, of the matrix represents a one-dimensional affine hyperplane. For a dependence from statement S_i to S_j (i.e. S_j depends on S_i), a valid hyperplane is one in which:

$$\phi_{S_j} - \phi_{S_i} \geq 0 \tag{1}$$

Note that equation 1 simply states that ϕ is valid if it schedules statement S_j to be executed after S_i. Valid hyperplanes can be identified with the help of Farkas lemma [47].

Lemma 2.1 (Affine form of Farkas Lemma). *Let $\mathcal{D}$ be a nonempty polyhedron defined by p affine inequalities*

$$\boldsymbol{a}_k \cdot \boldsymbol{x} + b_k \geq 0, 1 \leq k \leq p$$

Then an affine form $\psi(\boldsymbol{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination of the faces:

$$\psi(\boldsymbol{x}) \equiv \lambda_0 + \sum_k \lambda_k(\boldsymbol{a}_k \cdot \boldsymbol{x} + b_k), \lambda_k \geq 0$$

3. Related Work

Much work has been done in the field of iterative optimization. A strong argument for the benefits of iterative compilation is given by Kisuki et al. [27]. The authors show that the combinations of tile size and unroll factors that lead to good performance (i.e. within 3% of the global optimum) vary

wildly depending on the architecture. Optimizations suitable for one architecture, will not necessarily port to another. Bodin et al. [11] performed iterative compilation to optimize the selection of tile size, unroll factor, and array padding. Results were promising – achieving within 0.3% of the global optimum by visiting less than 0.25% of the search space.

Autotuning of full applications, as opposed to single functions, was performed by Tiwari et al. [56]. Profiling was first performed to identify computationally intensive loop nests and outline them into separate functions using the ROSE outliner [29]. Different code variants were then generated using CHiLL [44]. CHiLL is a loop transformation and code generation framework based on the polyhedral model [8]. It uses the polyhedral abstraction to represent loop iteration spaces and array accesses. Transformations such as loop permutation, tiling, unrolling, and skewing can be easily represented and composed using this framework. These transformation recipes can be used to generate a large number of code variants systematically. Similarly, Pouchet et al. [38] described a tractable optimization algorithm over the space of loop transformations. A recipe library was created by compiler experts to facilitate the tuning process. Finally, Active Harmony [52] was used to initiate the autotuning.

Active Harmony is an auto-tuning framework which allows programmers to define tunable parameters that describe the search space. A search algorithm can then be used to identify a near optimal configuration with relatively few evaluations. The Nelder-Mead simplex algorithm [33] was initially used by Tapus et al. [52]. Given an N dimensional search space, the Nelder-Mead algorithm maintains a set of $N + 1$ points forming a simplex. Each vertex corresponds to a different solution variant that is evaluated. Based on this evaluation, the simplex is either reflected, contracted, or expanded. Tiwari et al. [55] used the Parallel Rank Order (PRO) algorithm [51]. Parallel rank order is an extension of the Nelder-Mead simplex algorithm. The PRO algorithm is a parallelization of the Nelder-Mead algorithm in which N vertices are evaluated in parallel at each time step.

A number of domain specific auto-tuners have also emerged in recent years. ATLAS [41] is a popular library which tunes BLAS programs using a direct search method. Experiments showed that ATLAS could produce within 90% of the global optimum at a fraction of the search cost. PATUS [17] is framework for optimizing stencil programs using a domain specific language. It could be applied to multi-core CPUs as well as GPUs and optimized over loop unrolling, tiling, and vectorization on CPUs. SPIRAL [39] is

a tool for optimizing discrete signal processing applications. Algorithmic as well as code optimizations are applied. The search algorithms used include random search, dynamic programming, genetic algorithms and hill climbing [40]. Experiments show that SPIRAL is competitive with hand-tuned code. OpenTuner [6] is an extensible framework for empirical performance tuning based on heuristics.

Auto-tuning GPU specific applications has recently become popular due to the large number of GPU configuration parameters that can be tuned. In our previous work [37], ant colony optimization was used to identify the optimal CT reconstruction implementation. Given a graph of CUDA code segments defining different implementation variants, ants would assemble the segments to optimize for runtime performance. Results were competitive with hand-tuned code. An application independent GPU auto-tuner which optimizes thread block size and loop unroll factors was presented by Tillmann et al. [54]. Grauer-Gray et al. [24] tuned loop permutation, unrolling, and tiling on the GPU. Pragma directives were used to generate different program variants with the help of the HMPP compiler [20]. OpenACC [1] is a directive based programming model for GPUs and is similar to OpenMP [35]. Magni et al. [32] tuned the OpenACC parameters "gang" and "vector" for different input sizes. A random search was initially performed with a set of input sizes. Given a new input size, a nearest neighbor search was performed to narrow the search space. AlZayer et al. [5] studied different search algorithms for tuning of "gang" and "vector" parameters. They compared the results of random walk [45], simulated annealing [26], Nelder-Mead [33], and genetic algorithms [46] on 4^{th} and 8^{th} order stencils. Results showed that simulated annealing had the worst convergence time while genetic algorithms had the fastest convergence. Finally, Lift [49] provides a high-level functional data parallel language for GPU code optimization. It is defined primarily for OpenCL while we use CUDA. Other libraries and programming languages that aim to improve access of the GPU parallel execution facilities include Firepile [34], SafeGPU [28], and StreamIt [53].

Machine learning has also been used to improve iterative optimization quality and convergence time. Agokov et al. [4] used it to focus a random and genetic algorithm search to the most promising regions of the search space. Using a simple independent distribution model and a Markov model, they were able to obtain between 1.22x–1.27x speed-up in just a few evaluations. Ashouri et al. [7] introduced the COBAYN framework which is an auto-tuning framework that uses Bayesian networks to select the optimal se-

ALGORITHM 1: Ant Colony Optimization

```
1 Initialize pheromone values.;
2 while termination criteria not met do
3     ConstructSolutions()
4     ApplyLocalSearch()      % optional
5     UpdateTrails()
6 end
```

quence of compiler optimizations. Training was performed to learn statistical relations between application features and compiler optimizations. For new applications, iterative compilation was performed with the most promising optimizations as predicted by the Bayesian network. Chaimov et al. [15] kept a database to store the optimal variants of a program for different problem sizes. Decision trees were then used to predict the best variant given a new input size. Results for a matrix multiply kernel showed a 1.7x speedup compared to 1.3x for auto-tuning alone. Decision trees were also used to predict a good initial starting position for the PRO search algorithm – resulting in greatly improved convergence time. Lim et al [30] used static analysis to auto-tune GPU kernels without explicitly running them. Given a code variant, GPU occupancy and instruction throughput are used to estimate the performance of the kernel. Results showed that using instruction mix, in particular, was a good indicator of performance.

4. The Polyhedral Parallel Code Generator (PPCG)

The CA framework is built as an extension to the PPCG compiler. It takes C code as input and relies on PPCG to produce the polyhedral extraction. Our framework has three optimization levels which give increasing control to the ant colony optimization. The O1 optimization uses coding ants to select PPCG compiler options, including the scheduling algorithm, but lets the PPCG compiler actually perform the scheduling. The O2 optimization uses coding ants to select which statements get fused / fissioned at each level but relies on the PPCG compiler to compute the scheduling matrix. Finally, O3 gives complete control over fusion / fission and scheduling to the CA framework. The goal is to identify the optimal distribution of work between the PPCG compiler and the CA framework. For all three optimization levels, CA is responsible for selecting the remaining optimizations

(e.g. shared memory usage, tile size parameters, unroll factors, etc.)

As a motivating example we will look at the MVT benchmark in the PolyBench suite as an optimization case study (see figure 1). The MVT benchmark composes two matrix vector multiplications. It is a good candidate for GPU parallelism, since it has no dependences along the outermost loop. The PPCG generated code using the default options is shown in figure 1(b). It generates two, lightweight kernels that each parallelize over a single statement. This is not, however, the only possible configuration and overall performance can benefit from creating a single, coarser kernel.

Ant colony optimization has some advantages over other popular methods for iterative optimization such as parallel rank order and Nelder-Mead. First, it does not require the optimization search space to be hyper-rectangular. Second, framing the optimization space as a graph allows for the inclusion of meta-data to accelerate the optimization. For example, performing a skewing transformation will only hurt performance for many applications. By separating skewing and interchange transformations to be rooted at different nodes in the graph, as shown in Figure 6, the ant colony optimization algorithm can determine whether skewing or interchange, in general, produces better results.

5. Coding Ants

As mentioned, our Coding Ants parallel code optimization framework makes use of Ant colony optimization (ACO). ACO is an optimization approach typically used for solving NP hard problems. It mimics the way ants find a path to a food source in nature. Ants initially perform a random search. When an ant finds food, it will travel back to the colony while laying down a pheromone trail. This acts as a type of indirect communication to help other ants find the food source. The pheromone trail, however, evaporates over time. Given multiple paths to a food source, the pheromone on the shortest path will have less time to evaporate before being reinforced by the next ant. Eventually, all ants will converge to a single, shortest path.

Algorithm 1, from [9], shows the algorithmic skeleton for ACO algorithms. For each iteration of the *while* loop, a number of ants construct solutions. Solutions are constructed from a graph where each node represents a component of the solution. A path through the graph represents a complete solution which can be evaluated. Ants construct solutions probabilistically. Specifically, ant k at node i chooses to move to node j according to:

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
   x1[i] += A[i][j] * y1[j]

for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
   x2[i] += A[j][i] * y2[j]
```

(a) MVT benchmark C code

```
// Kernel 1 ...
tx = threadIdx.x
bx = 32 * blockIdx.x
for (it = 0; it < N; it += 32)
  shared_y2[tx] = y2[tx + it]
  syncthreads()
  for (i = 0; i < 32; ++i)
   x2[tx + bx] += A[i + it][tx + bx] * shared_y2[i]
  syncthreads()

// Kernel 2 ...
tx = threadIdx.x
bx = 32 * blockIdx.x
for (it = 0; it < N; it += 32)
  for (i = 0; i < 32; ++i)
   shared_A[i][tx] = A[it + bx][tx + i]
  shared_y1[tx] = y1[tx + it]
  syncthreads()
  for (i = 0; i < 32; ++i)
   x1[tx + bx] += shared_A[tx][i] * shared_y1[i]
  syncthreads()
```

(b) MVT benchmark PPCG generated code

Figure 1: MVT PolyBench benchmark C code (a) and default PPCG generated code(b). The PPCG code generates a CUDA kernel for each statement and makes heavy use of shared memory.

$$p_{ij}^k = \frac{\tau_{ij}^\alpha \eta_{ij}^\beta}{\sum_{l \in N_i^k} \tau_{il}^\alpha \eta_{il}^\beta} \tag{2}$$

where τ_{ij} is the pheromone amount on the edge from i to j and η_{ij} is some *a priori* desirability for moving from node i to node j. α and β control the pheromone and *a priori* influence respectively. N_i^k is the set of all possible transitions ant k can take from node i.

After a solution is constructed, the pheromone values are updated according to:

$$\tau_{ij} = (1 - \rho) \cdot \tau_{ij} + \rho \cdot \sum_{s \in S_{upd}} w_s \cdot F(s) \tag{3}$$

where S_{upd} is the set of ant solutions that will be updated. w_s is a weight and $F(s)$ represents the quality of solution s. ρ is the evaporation coefficient and it determines how quickly the pheromone evaporates.

Ant system (AS) [19], was the first ACO algorithm developed. It was used to solve the traveling salesman problem. Although AS produced encouraging results, it was not competitive with state of the art algorithms. Algorithmic improvements have been applied and a number of ACO algorithms have since been developed. Elitist AS [18] was the first extension to the AS algorithm. At the end of each iteration, in addition to the ants updating the pheromone values along their respective paths, the best-so-far path was updated as well. This helps to bias the ants' path selection to the best known solution. Rank-based AS [14] was another extension over the AS algorithm. Instead of performing the pheromone update on all ants in the iteration, only the best $m - 1$ ants were updated along with the best-so-far solution. The weights of each solution were set based on its rank such that $w_s = m - r_s$, where r_s is the rank of solution s.

The Max-Min Ant System (MMAS) [50] is among the best performing ACO variants. In MMAS, the pheromone trail is updated based on a single ant. Depending on the iteration, either the best ant or the best-so-far trail is updated. In the earlier iterations, the iteration best path is updated more often, while in the later iterations the best-so-far path is updated more often. This helps to prevent premature convergence to the globally best solution. Upper and lower bounds on the amount of pheromone are also set. A constant lower bound of $\tau_{min} > 0$ is used while the upper bound is set to $F(s_{bs})/\rho$. By setting bounds on the pheromone values, Stutzle and Hoos [50] were able to prove that the global optimum could be found in a finite number of iterations.

Table 1: List of Optimizations

Optimization	Level	Description
Scheduling Algorithm	O1	Pluto or Feautrier Scheduling algorithm
Serialize SCCs	O1	CUDA kernel for each SCC
Max Band Depth	O1	Maximize the depth of permutable bands
Stmt. Order	O2, O3	Set the lexicographic statement order
Partition Stmts.	O2, O3	Fuse / Fission loops and kernels
Schedule Matrix	O3	Construct the scheduling matrix
Reduction	O1, O2, O3	Use parallel reduction
Memory Placement	O1, O2, O3	Store arrays in texture or global memory
Shared Memory	O1, O2, O3	Toggle use of shared memory per kernel
Cache Config.	O1, O2, O3	Set the cudaFuncSetCacheConfig variable
Outer Tile Size	O1, O2, O3	Set the tile size for the parallel dims.
Thread Block Size	O1, O2, O3	Set the thread block sizes
Unroll Factors	O1, O2, O3	Set the unroll factors for intra-kernel loops
Inner Tile Size	O1, O2, O3	Set tile sizes for intra-kernel permutable bands
Atomic Store	O1, O2, O3	Replace Stores with atomic stores

The hyper-cube framework (HCF) [10] is an extension to ACO algorithms that replaces the weight function in equation 3 with $(\sum_{s' \in S_{upd}} F(s'))^{-1}$. This bounds the pheromone values to between 0 and 1. It was proven that for unconstrained optimization problems, the expected solution quality increased monotonically for each iteration. This also has some practical benefits in that the objective functions are automatically scaled. Additionally, the upper bound in the MMAS implementation does not need to be recomputed when a new global best solution is found.

5.1. Optimization Space

Table 1 shows a list of optimizations that are considered by the CA framework. There are three optimization levels. O1 optimizes the selection of compiler options that are exposed by the PPCG compiler. O2 optimizes the partitioning of statements into CUDA kernels and intra-kernel loop nests. The scheduling matrix, however, is selected by the PPCG compiler. For O3, the scheduling matrix is also selected by the coding ants optimization. A majority of the optimizations considered are used for all three optimization levels. These optimizations are applied after the schedule has been set and

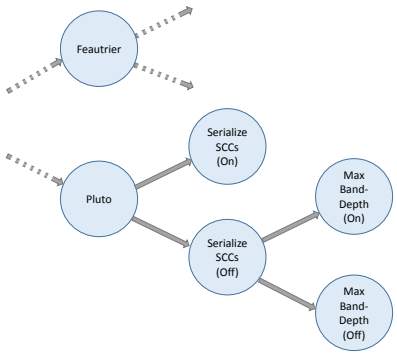


Figure 2: The O1 optimization subgraph. Note that the subtree rooted at the "Feautrier" node is omitted for brevity and is a clone of the subtree rooted at the "Pluto" node.

the number of kernels has been determined.

5.1.1. O1

The O1 optimization for the coding ants framework gives maximal control of the scheduling to the PPCG compiler. Its effect on scheduling is only in the selection of algorithms and various compiler options available through the PPCG compiler. Figure 2 shows the optimization subgraph for the O1 optimization level. The first choice is in the selection of the scheduling algorithm. The options are either the Pluto algorithm [12] or the Feautrier algorithm [22][23]. The Pluto algorithm is able to obtain outer loop parallelism and permutable bands. The code generated with this algorithm is more likely to contain CUDA kernels that are relatively coarse (i.e. they have deep intra-kernel loops and perform a lot of work). It is also more likely to tile intra-kernel loops. The Feautrier algorithm is more adept at exposing inner loop parallelism. Code generated by the Feautrier algorithm is more likely to contain relatively lightweight CUDA kernels that are nested within sequential *for* loops.

The serialize SCCs compiler option separates each strongly connected component (SCC) into its own CUDA kernel. Statements will only be placed in the same kernel if they belong to the same strongly connected component. The effect of the max band depth option is to maximize the width of the permutable bands. Two SCCs are only fused if they do not prevent permutability or parallelism. The maximize band depth option is only applicable if the serialize SCCs option has not been selected. This is because the serialize SCCs option prevents any fusion from happening and so the band depth is

```
// Kernel 1 ...
tid = threadIdx.x + 32 * blockIdx.x
for (it = 0; it < N; it += 16)
  shared_y2[threadIdx.x] = tex1D(y2, threadIdx.x + it)
  syncthreads()
  for (i = 0; i < 16; ++i)
   x2[tid] += tex2D(A, i + it, tid) * shared_y2[i]
  syncthreads()

// Kernel 2 ...
tid = threadIdx.x + 32 * blockIdx.x
for (it = 0; it < N; it += 32)
  for (i = 0; i < 32; ++i)
  shared_y1[threadIdx.x] = y1[threadIdx.x + it]
  syncthreads()
  for (i = 0; i < 32; ++i)
   x1[tid] += tex2D(A, threadIdx.x, i) * shared_y1[i]
  syncthreads()
```

Figure 3: CUDA code generated by the O1 optimization. The major difference from the PPCG generated code is the use of texture memory to store arrays `A` and `y2`.

trivially maximized.

Returning to our motivating example, figure 3 shows the result of applying the O1 optimization on the MVT benchmark. The structure of the code is similar to that of the default PPCG code of figure 1(b). In this benchmark, however, performance is improved by the use of texture memory. This is especially beneficial for kernel two, as the access pattern on array `A` is uncoalesced – leading to long latencies when retrieving data.

5.1.2. O2

The O2 optimization gives control over the scheduling matrix to the PPCG compiler but lets the CA framework select the optimal partitioning of statements to kernels and loop nests. There are three steps the framework takes for partitioning statements. First, a lexicographic statement ordering is selected. This is done by first topologically sorting the SCC dependence graph. Any valid topological ordering may be selected. Next, the number of partitions is selected. A partition represents a single loop nest. The number

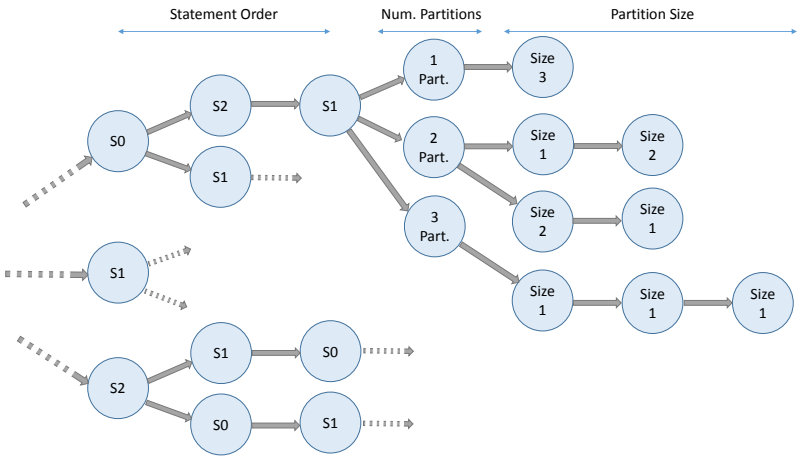


Figure 4: The O2 optimization subgraph for a program with three statements and no inter-statement dependences. Note that only the top branch of the graph is shown in full and dashed edges represent clones of the top branch. Each partition "size" node represents the number of SCCs in a partition.

of valid partitions ranges from 1 to N, where N is the number of SCCs. If there is 1 partition then there is max fusion (i.e every SCC will be nested under the same loop). Conversely, N partitions represents max fission (i.e. every SCC will be nested under a separate loop). Finally, the partition sizes are selected. The size of a partition represents the number of SCCs it contains. Valid partition sizes vary depending on the number of partitions. If there is a single partition, then only one valid partition size exists (i.e. N). If there are N partitions, however, then each partition has a size of 1. If the number of partitions is between 1 and N, then there are multiple options for selecting the partition sizes of each partition. An example of this is shown in figure 4.

Figure 5 shows the result of applying the O2 optimization on the MVT benchmark. The CA framework has chosen to fuse the two statements into a single, coarser kernel. This reduces overhead associated with launching CUDA kernels as well as improves cache reuse along array `A`. A second level of parallelism is also exposed with this optimization. Note that in figure 1(a), the innermost loop is a prime candidate for parallel reduction. This is enabled as an option in the CA framework and utilizes a combination of shuffle instructions and atomic operations. More detail about how reduction parallelism is exposed can be seen in Section 5.1.4.

```
// Kernel 1 ...
tx = threadIdx.x, ty = threadIdx.y
bx = 32 * blockIdx.x, by = 32 * blockIdx.y
for (i = tx; i < 32; i += 16)
  reduce(x2[bx + i], tex2D(A, bx + i, ty + by) * tex1D(y2, ty + by))
  reduce(x1[bx + i], tex2D(A, ty + by, bx + i) * y1[ty + by])
```

Figure 5: CUDA code generated by the O2 optimization. Statements have been fused into a single kernel. The kernel also exposes an additional level of parallelism via parallel reduction.

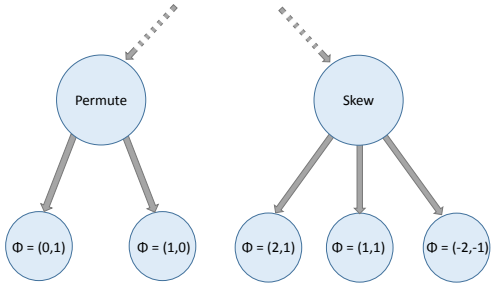


Figure 6: The O3 optimization subgraph. Note that the hyperplanes under the "Permute" node are all rows of the identity matrix.

5.1.3. O3

The O3 optimization gives complete control of scheduling to the CA framework. The framework is responsible for selecting a scheduling hyperplane, ϕ, for each statement as well as defining the partitions. The hyperplane is selected by CA from a set of valid hyperplanes for the current scheduling level and SCC. Schedules are created by setting each of the coefficients of ϕ between $[-2, 2]$ and checking if the schedule is valid with respect to all of the intra-SCC dependences. Given a candidate schedule, ϕ, for each dependence edge, e, between statements S_i and S_j in the SCC, we check:

$$\mathcal{P}_e \subseteq \phi_{S_j} - \phi_{S_i} \tag{4}$$

where $\mathcal{P}_e$ is the dependence polyhedron. Equation 4 guarantees that the schedule does not violate any dependences and follows directly from Farkas Lemma. If the candidate schedule satisfies equation 4 then it is added to a set of valid schedules that will be considered by the coding ants framework.

The schedule is selected prior to partitioning. In general, scheduling will have more impact on performance than the partitioning. An appropriate schedule can expose parallelism and permutability; two aspects that are particularly important for GPU acceleration. Additionally, it is simpler to find a partition that conforms to a selected schedule than to select a valid schedule given a partitioning of statements. In general, it may not be possible to produce a valid schedule where two statements are fused under the same loop nest. It is, however, always possible to produce a valid partitioning through maximum fission of the SCCs.

Figure 6 shows the optimization graph for selecting the scheduling hyperplane. The selected hyperplanes are then organized into a scheduling matrix. Note that we separate the valid schedules into permutation and skewing schedules. A permutation schedule is a matrix that consists of rows of the identity matrix. It results in code which simply permutes the order of loops in the input code. A skewing schedule consists of a matrix in which any row contains a two or more non-zero coefficients. It usually results in more complex code generation but can be useful for obtaining parallelism and permutability if none exists in the input code. Separating the schedules into these two categories is useful because there are typically more ways to skew than to permute. This introduces a bias in the framework for selecting schedules that skew. Furthermore, skewing when it is not beneficial is usually harmful to performance. It can remove existing parallelism and

```
// Kernel 1 ...
tx = threadIdx.x
bx = 64 * blockIdx.x
for (it = 0; it < N; it += 8)
  for (i = 0; i < 8; ++i)
   x2[bx + i] += tex2D(A, tx + bx, it + i) * tex1D(y2, it + i)
   x1[bx + i] += tex2D(A, it + i, tx + bx) * tex1D(y1, it + i)
```

Figure 7: CUDA code generated by the O3 optimization. Statements have been fused into a single kernel but the parallel reduction has been lost.

permutability in the input code. By adding an additional layer to separate skewing from permutation schedules, the initial probability of selecting either remains 50%.

Figure 7 shows the result of applying the O3 optimization level to the MVT benchmark. Although the statements have been fused and texture memory is still being used, reduction level parallelism has not been exposed. This is possibly due to the large increase in the search space caused by evaluating the different candidate schedules. As is seen in Section 7, the O3 optimization level performs worse than the O2 level for this benchmark, largely because it missed the reduction level parallelism.

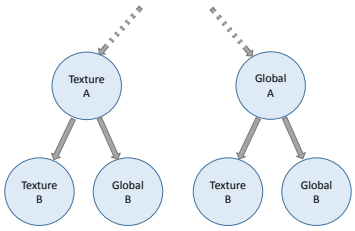
5.1.4. Reduction

Parallel reduction is a powerful technique that exposes some amount of parallelism. A loop is a candidate for parallel reduction if (1) the only dependence within the loop is a write-after-write (WAW) dependence (2) each iteration of the loop writes to the same memory location, and (3) the arithmetic operator is associative [43]. Although parallel reduction has been implemented in CUDA [25], it is not often utilized in automatic GPU code generation. As a part of the coding ants framework, intra-kernel parallel reduction is implemented.

Intra-kernel parallel reduction is implemented based on the algorithm by Luitjens et al. [31]. This involves using shuffle instructions to perform a warp level reduction to reduce to a single value per warp, followed by an atomic add. Atomics have traditionally been a very slow operation in parallel programming. The Kepler architecture of NVIDIA GPUs, however, has redesigned atomic operations to be asynchronous. With the newer generation of GPUs, atomics only introduce bottlenecks if there is a lot of contention

```
for (i = 0; i < N; ++i)
  for (j = 0; j < N; ++j)
   C[i][j] = A[i][j] * B[j][i]
```

(a)



(b)

Figure 8: Source code with a poor access pattern on `B` (a) and the corresponding optimization graph (b).

on a single memory location. By performing the warp level reduction first, the contention is reduced by a factor of 32 (i.e. the warp size). Additionally, when the write operation of a parallel reduction is to a location in shared memory, the atomic contention is limited within the thread block.

The inclusion of the warp level reduction creates some additional constraints for the parallel reduction detection. In addition to the three criteria mentioned earlier, the write array cannot be indexed by the x dimension of the thread block. Each thread in the warp is viewed as an iteration of the reduction loop. If the write array is indexed by the x dimension then each thread in the warp no longer represents a separate iteration of the same reduction loop. Each thread writes to a different memory location and so cannot be reduced to a single value per warp.

The selection of parallel reduction in the coding ants framework is relatively straightforward. Parallel reduction is toggled on or off for each partition and is only possible if every statement in the partition can be reduced. The partition can be selected either through CA (via the O2 optimization level) or by the PPCG compiler (through the O1 optimization level). If parallel reduction is enabled, then an additional level of parallelism is applied to the CUDA kernel. This creates a fifth criteria for enabling parallel reduction. There must be at most two parallel dimensions found so far during scheduling. This is because CUDA allows for up to three dimensions of parallelism. At least one dimension needs to be reserved for parallel reduction.

5.1.5. Memory Placement

GPUs have many different regions in which to store data (e.g. global memory, texture memory, etc.). These different regions contain caches that have different desirable access patterns. For example, global memory is similar to traditional DRAM in which accessing data sequentially is the best access pattern. Texture memory, however, is organized with a space-filling curve [2], and so is optimized for 2D spatial locality. This is particularly useful for storing matrices where it is not possible to access data sequentially. Figure 8(a) shows a simple example where texture memory can be useful. Note that it is not possible to obtain spatial locality for both the `A` and `B` matrices simultaneously.

The CA framework optimizes the placement of arrays into either global or texture memory. All arrays can be placed into global memory, but texture memory is only available to read-only arrays. Figure 8(b) shows the optimization graph for memory placement. Note that the optimization is not unconstrained. Placing array A in texture memory affects the likelihood of every other array also being placed in texture memory. This is to account for texture cache contention. Placing every array in texture memory may cause too many cache capacity misses to be beneficial.

5.1.6. Shared Memory + Cache Configuration

Shared memory acts as a type of user controlled cache. It is often useful to use shared memory in cases where an un-coalesced access pattern is required. Data is loaded into shared memory in a coalesced way, and the remainder of the computation is performed using the data in shared memory. Shared memory usage was incorporated into the PPCG compiler. Shared memory usage can be toggled on and off through a compiler option. It is, however, a global option and when enabled, uses shared memory in all kernels whenever possible. The CA framework implements shared memory at finer granularity and additionally sets the CUDA cache configuration for each kernel.

There are three CUDA cache configurations – increase the amount of shared memory, no preference, and increase the amount of L1 cache. Since they share a physical location, the configurations allow for a trade-off between L1 cache and shared memory size. When shared memory is enabled, all three configuration options are available to the traversing ant. When it is disabled, however, only the "no preference" and "prefer L1 cache" options are available. This is a straightforward implementation decision because it does not make sense to increase the size of shared memory if it is not being used.

5.1.7. Outer Tile + Thread Block Size

The outer tile size (OTS) refers to the tile factors of the parallel dimensions. The OTS determines how much work each thread block will perform. The thread block size (TBS) refers to the thread block dimensions of the CUDA kernel. The OTS and TBS together determine the thread granularity (i.e. the amount of work per thread). For example, if the OTS is 64 and the TBS is 32, then each thread will have a granularity of two. Setting the TBS equal to the OTS, however, results in a thread granularity of one. In general, the granularity can be defined as OTS/TBS.

There are a few constraints on the selection of tile and thread block sizes. First, the thread block size must be less than or equal to the corresponding outer tile dimension. A TBS greater than the OTS would imply a thread granularity that is less than one. Second, the total OTS is limited to 1024. This is due to the GPU constraint that no thread block can contain more than 1024 threads. Finally, a thread block cannot contain less than 32 threads, as it would lead to unutilized threads in a warp. For both the OTS and TBS, the possible values are 32, 64, 128, 256, or 512 for the x dimension; 4, 8, 16, or 32 for the y dimension; and 2, 4, 8, or 16 for the z dimension.

5.1.8. Unroll Factors + Inner Tile Size

The CA framework selects optimal unroll factors and tile sizes for intra-kernel loops. Unrolling loops has several advantages. Before each iteration of the loop, a check must be made to determine if the thread should exit the loop. Loop unrolling reduces the number of these checks. Unrolling can also improve register usage and expose instruction level parallelism. Tiling is a combination of strip-mine plus loop interchange and is commonly used to improve data locality. A traversing ant may select unroll factors of 1 (i.e. no unroll), 2, 4, or 8. The possible tile sizes are 1 (i.e. no tile), 4, 8, 16, 32, or 64. The PPCG compiler provides mechanisms to apply both of these transformations.

Similar to the selection of thread block sizes, we place additional constraints on the selection of unroll factors and tile sizes. Specifically, the tile sizes must be greater than the unroll factors. As mentioned by Chen [16], selecting a tile size less than the unroll factor will not yield much improvement, since the data in the tiles will already be stored in registers. Selecting optimal unroll factors is particularly important since it has important implications on occupancy. Unrolling loops will usually increase the register pressure and may cause a drop in occupancy. This may still be beneficial,

however, as the benefits to register usage and instruction level parallelism may improve the overall runtime.

5.1.9. Atomic Store

As mentioned earlier, NVIDIA GPUs have greatly improved the atomics since the Kepler architecture. Atomics are now implemented asynchronously and is only slow when there is a lot of contention. If there is no contention, however, using atomics can still be used for asynchronous stores. This type of optimization was first used in our earlier work [36] and typically yields a modest speed-up. The use of atomic stores is considered in the CA framework. Specifically, atomic store is toggled on or off for each kernel. When enabled, all stores to global memory will be replaced with the *atomicExch* function.

5.2. Optimization Graph

Since the optimization space is so large, the graph is not fully constructed prior to running the ant colony optimization algorithm. Instead, it is constructed throughout the traversal. A node in the graph is only created when an ant chooses a previously untouched edge to traverse. The destination node of the edge is initialized and appropriate edge pointers are updated. A part of the initialization is to identify the number of outgoing edges. This can be computed based on the type of optimization the node represents. For example, if the node represents an optimization that is toggled on or off (e.g. shared memory, atomic store, etc.), the node will have two outgoing edges – one edge to enable the optimization and one edge to disable the optimization. The graph can be dynamically constructed in this way because the ant only needs to know the number of outgoing edges to make a decision according to equation 2.

Like previous work [52][29], the CA framework uses the polyhedral model to apply certain transformations (e.g. loop permutation, fusion, fission, etc.). Unlike previous work, however, the set of transformations to consider are not selected prior to running the iterative optimizer. The CA framework is tightly integrated with the PPCG compiler and these transformations are selected during scheduling. This has the advantage that no additional analysis is needed to identify legal transformations. It also gives CA insight into the full space of valid transformations. Algorithm 2 shows the modifications made to the Pluto algorithm [12] to allow the CA framework to select transformations. Note that H_S is the transformation matrix for statement S and $H_S^{\perp}$ is the

orthogonal subspace to the H_S. The call to the *ConstructSolution* function in line 17 triggers the traversal of O3 optimization subgraph of figure 6. Similarly, calling *ConstructSolution* in line 24 triggers the traversal of the O2 optimization graph of figure 4 to select a partitioning of statements. Note that cutting dependences is synonymous with partitioning.

One advantage of framing the optimization space via a graph is the ability to insert domain expertise. Optimizations that are expected to have a significant impact on performance (e.g. scheduling matrix, partitioning, etc.) are placed toward the top of the optimization graph. Nodes near the root of the graph will be evaluated more thoroughly because ants are more likely to select them and thus, better communicate their impact of the optimization on performance. Another advantage of using a graph is the ability to add nodes that correspond to meta-data. These are nodes that do not directly map to an optimization, but can help guide ants toward an optimal solution. An example of this meta-data is seen in the selection of scheduling matrices in Section 5.1.3. The use of a graph allows us to divide the schedules into skewing and permutation transformations.

The use of a graph as the optimization space also allows for the pruning of likely bad solutions. Pruning the search space of known bad solutions has been known to significantly improve solution quality and reduce the search time [16]. The optimization graph is pruned in several ways. As seen in the previous section, we prune the thread block size and the intra-kernel tile sizes. The possible thread block sizes is dependent on the selection of the outer tile sizes. Similarly, the intra-kernel tile sizes are pruned based on the selection of unroll factors.

The possible partitions are also pruned to ensure at least two dimensions of parallelism when it is available. A constraint is added such that two SCCs with different levels of parallelism cannot be fused at the outermost dimension. This prevents parallelism from being disabled due to fusion. Additionally, if multiple SCCs are fused at the outermost dimension, and they have parallelism at the next dimension, then they must be also be fused at the second dimension. If SCCs are fused at the outermost dimension, then they will be contained in a single CUDA kernel. Fissioning at the second dimension, will force the kernel to launch with a one-dimensional thread block. Thus, even though parallelism exists at the second dimension, the generated code will be unable to exploit this parallelism. When the O3 optimization level is enabled, pruning also takes place to ensure that the schedules selected contain sufficient parallelism. This is done by selecting

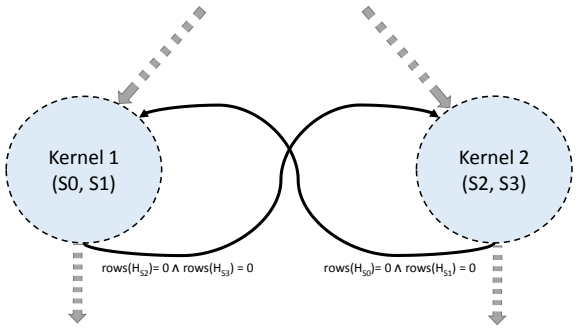


Figure 9: Optimization graph for two kernels with no dependences. Kernel 1 contains statements S0 and S1, while kernel 2 contains statements S2 and S3. Semantic edges are shown in black and are traversed whenever the condition, shown as the edge label, is true. Note that the dashed circles represent intra-kernel optimization subgraphs for partitioning statements into intra-kernel loops.

candidate schedules that not only satisfy equation 4 but also satisfy $\mathcal{P}_e \supseteq \phi_{S_j} - \phi_{S_i}$ for the outer two dimensions.

Optimization graphs that are used for ant colony optimization are typically directed acyclic graphs (DAG). Ants that traverse DAGs will always terminate. The requirement that optimization graph is a DAG presents a problem. Consider the case where there are two CUDA kernels that are completely independent (i.e. there exists no dependence between kernel 1 and kernel 2). In a DAG optimization graph, there exists two paths – one where kernel 1 is executed first, followed by kernel 2 and a second path where kernel 2 is executed first followed by kernel 1. An acyclic optimization graph will contain multiple copies of the same kernel. This is problematic, however, as intra-kernel optimizations (e.g. tiling, unrolling, thread block sizes, etc.) will only effect the performance of a single kernel. Multiple copies of the kernel in the optimization graph will hinder the ants' ability to communicate about the value of the different solution components.

The CA framework eliminates problems associated with multiple copies by adding cycles. Figure 9 illustrates how cycles are added to ensure that lexicographic ordering does not result in unnecessary duplicates of parts of the optimization graph. Cycles are introduced by adding a new type of edge, called semantic edges, to the optimization graph. These semantic edges differ from the rest of the edges in the graph in that they are traversed by an ant when some deterministic criteria is satisfied. Specifically, the statements

being scheduled and the current scheduling level are used to determine which semantic edge should be traversed. The semantic edge from kernel 1 to kernel 2 in figure 9, for example, is traversed when the number of chosen hyperplanes for S2 and S3 is zero (i.e. the statements in kernel 2 have not yet been scheduled). Although figure 9 shows how cycles prevent duplicate kernels in the optimization graph, the same process is used to prevent duplicates when partitioning statements into intra-kernel loops as well.

6. ACO Extensions

The coding ants framework applies the Max-Min Ant System (MMAS) with hypercube formulation (HCF). Recall from Section 5 that the effect of applying the HCF is that the weight in equation 3 becomes $(\sum_{s' \in S_{upd}} F(s'))^{-1}$. Additionally, the MMAS updates the pheromones based on a single ant (i.e. either the iteration best ant or the best-so-far ant). Applying the HCF with the MMAS simplifies the update function in equation 3 to:

$$\tau = (1 - \rho) \cdot \tau + \rho \cdot 1 \tag{5}$$

6.1. Per Kernel Updates

Extensions to this algorithm are also applied to make it more amenable to GPU performance optimization. In addition to updating the pheromone values with the an ant based on the global runtime performance, pheromone values are also updated based on the per kernel timings. Some solutions may have a poor global runtime performance because of bottlenecks in a single kernel. The optimizations selected for other kernels may, however, be profitable. The CA framework keeps track of the best ant per kernel and updates each of the intra-kernel optimization subgraphs using equation 5 at the end of each iteration. The benefit of this extension is that optimizations selected for one kernel will not be unfairly penalized because of optimizations selected for another kernel.

6.2. Performance Metrics

Another extension to the ACO algorithm is the use of performance metrics. Performance metrics can be used to communicate additional information between the ants. Instead of communicating that using texture memory is desirable, for example, it can communicate that it is desirable because it leads to a high cache hit rate. Conversely, ants can also communicate that

parallel reduction, for example, leads to a 50% drop in occupancy, and so it should be avoided. The idea is to encourage ants to select optimizations which address current performance bottlenecks, as it has the best chance of improving performance.

The performance metrics that are gathered from each evaluation are shown in table 2. After the evaluation of an ant, the performance metrics are gathered using *nvprof* [3] and are normalized between 0 and 1 – where 1 is the theoretical peak. The value of each performance metric is then stored at each edge along the ant's path along with the standard deviation. The metric values at each edge are then used to compute the *a priori* desirability, η, from equation 2. The *eta* value is computed by:

$$\eta_{ij} = 1 + \frac{1}{|M_{ij}|} \cdot \sum_{\substack{m_{ij} \in M_{ij} \\ m_{sb} \in M_{sb}}} \frac{(m_{ij} - m_{sb})}{\sigma_{m_{ij}} + 1} \tag{6}$$

where M_{ij} is the set of performance metrics from table 2 accumulated on the edge from node i to node j. M_{sb} is the set of performance metrics gathered from the best-so-far solution. $\sigma_{m_{ij}}$ is the standard deviation of metric m on the edge ij.

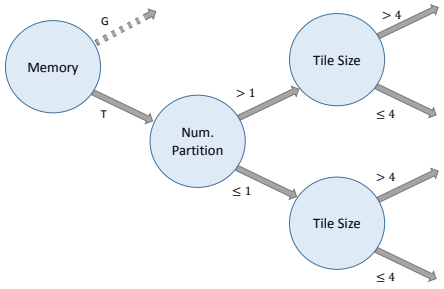
The effect of equation 6 is to weight edges more heavily which address the performance bottlenecks of the best-so-far solution. The performance metric, m_{ij}, acts as a prediction. If edge ij is traversed, we expect that metric m will have a value that is close to m_{ij}. The influence of each performance metric is controlled by the standard deviation of the metrics on edge ij. If the standard deviation is low, then m_{ij} is a good predictor of performance.

6.3. Regression Tree

The last extension the coding ants framework makes to the traditional ACO algorithm is the utilization of regression trees [13]. Regression trees are similar to decision trees [42], except instead of classification, regression trees model piecewise constant functions. Given a number of samples, the regression tree is built by recursively splitting the samples until some criterion is met (e.g. the number of samples is less than some threshold). Each node in the resulting tree represents a split. Splits are chosen such that the variance of the children is minimized. Regression trees can be used to predict the performance of applying a set of optimizations. The CA framework uses regression trees to guide ants' optimization choices.

Sample	# Part.	Mem.	Tile	
P0	2	T	4	
P1	2	T	8	
P2	1	T	4	
P3	1	T	8	
P4	2	G	4	
P5	2	G	8	
P6	1	G	4	
P7	1	G	8	9s

(a)



(b)

Figure 10: Regression tree (b) constructed from sample points (a) corresponding to optimizations selected by 8 ants. A subset of the optimizations are shown for brevity and include number of partitions, memory placement (i.e. T for texture and G for global memory), and tile size. Score is the runtime measured in seconds. Note that even though partitioning is performed before memory placement in the optimization graph, the regression tree places memory placement at the root because of its greater performance impact.

A regression tree is constructed before each iteration from the solutions that have been discovered from all prior iterations. These solutions act as samples of the optimization space, where the different optimizations act as the attributes. Figure 10 shows an example of a regression tree constructed from previous solutions. Splits are represented through linear inequalites. For the O2 and O3 optimization levels, the regression tree is constructed from a subset of the optimizations. Only partitioning at the outermost level is considered with the O2 optimization. This lets the regression tree determine the partitioning of statements to CUDA kernels, but it is not concerned with intra-kernel loops. When O3 is enabled, the regression tree only considers whether a statement's schedule is a skewing or a permutation hyperplane. The termination criteria is based on the number of samples. Specifically, construction terminates if the number of samples is ≤ 2. For each level of the recursion, a split is chosen which minimizes the variance of the global runtime. This effectively segments the optimization space into regions that have promising performance.

The regression tree is viewed as an extension to the optimization graph. After it is constructed, the edges of the regression tree are updated with pheromones according to the original ant system algorithm [19]. This is done by identifying the path through the newly constructed regression tree

that an ant would have taken based on its selected optimizations. For each new ant in the next iteration, the regression tree is traversed prior to traversal of the optimization graph. The nodes correspond to splits and are recorded by the traversing ant. The splits act as filters. Throughout the traversal of the optimization graph, the ant must only select paths which respect the splits that have been traversed in the regression tree. For example, if the ant traverses the bottom path of the "Memory" node in figure 10(b), then the ant must choose to use texture memory when it reaches the appropriate edge in the optimization graph. This type of filtering is easy to incorporate in ACO algorithms. It corresponds to obstacles being placed on some paths, making them impassable.

There are several advantages to using a regression tree with ACO. First, splits toward the top of the regression tree correspond to distinguishing optimizations specific to the program being optimized. For example, loop unrolling optimizations are selected toward the end of the optimization graph. This implies that loop unrolling will have little impact on performance. Ants will have less communication toward the end of the graph because of the many different paths that are available in the optimization graph. If this assumption is wrong, however, the construction of the regression tree will place splits relating to loop unrolling at the root of the tree. Ants will make this decision first and so more effectively communicate the benefits of loop unrolling.

Another advantage of the regression tree is that it can help focus ants to search specific subspaces. These subspaces are not selected prior to optimization, but rather they are discovered by sampling the optimization space. For each iteration, the CA framework directs half of the ants to search the subspace of the best-so-far solution. This allows for a type of local search, in which the ants are searching near the best-so-far solution by applying the most crucial optimizations, and searching among the others.

7. Experiments

We performed experiments to evaluate our approach (all code is available on GitHub [1]) using the PolyBench suite version 4.2.1 [2]. We compare the

[1] https://github.com/epapenhausen/CodingAnts

[2] http://web.cse.ohio-state.edu/ pouchet.2/software/polybench

performance of the Coding Ants framework to that of the performance generated by the PPCG compiler version 0.05 alone as well as a random search. In all cases CUDA code was generated and compiled using NVCC version 8.0. The performance was evaluated on two GPUs – the NVIDIA GeForce GT 755m and NVIDIA GeForce GTX 1070.

The NVIDIA GeForce GT 755m has a compute capability 3.0 (i.e. Kepler architecture). It contains 2 multiprocessors with 192 CUDA cores each, operating at 1.02 GHz. It contains 65,536 registers and 48KB of shared memory per thread block and has 256KB of L2 cache. The NVIDIA GeForce GTX 1070 has a compute capability of 6.1 (i.e. Pascal architecture). It contains 15 multiprocessors with 128 cores operating at 1.75 GHz. It contain 65,536 registers and 48KB of shared memory per thread block and has 2MB of L2 cache.

We compare the performance of the three CA optimization levels with the default PPCG strategy and a random search through the graph. The CA optimizations were run with 10 ants for 5 iterations – for a total of 50 evaluations. The α and β values from equation 2 were both set to 1 and the evaporation coefficient, ρ, was set to 0.1. The performance of each evaluation was determined as the mean execution time of 10 executions, and so the time required for an optimization was 10 ants $\times$ 5 iterations $\times$ 10 executions $\times$ code runtime. The default PPCG strategy uses the Pluto algorithm with the maximize band depth option enabled. Shared memory is also used whenever possible. The random search evaluates the performance of 50 random paths through the O2 optimization graph.

Figure 12 shows the performance of the CA framework for the GT 755m GPU. An average speed-up of 2.85, 3.02 and 2.44 was obtained for the O1, O2, and O3 optimization levels respectively over the default PPCG generated code. The random search only achieved an average speed-up of 2.03. There were several benchmarks in which CA did not improve over the PPCG compiler (*2mm*, *3mm*, *gemm* and *syrk*). These benchmarks required careful tuning of thread block and tile sizes to obtain good performance. Furthermore, there did not appear to be a strong correlation between performance and distance to the global optimum for these examples. This makes the search process more difficult. In several benchmarks, a dramatic performance improvement was gained largely because of the new optimizations we have included. The *doitgen* benchmark, for example, saw a 2x performance improvement just from using texture memory. This further improved to 3x by including parallel reduction as well.

Figure 13 shows the performance on the GTX 1070 GPU. Average speed-ups of 1.58, 1.56, and 1.3 were obtained for the O1, O2, and O3 optimization levels. The random search achieved a speed-up of 1.4. The O1 optimization performed slightly better than the O2 optimization on this GPU and even outperformed the PPCG compiler on the *2mm*, *gemm*, and *syrk* benchmarks. Interestingly, the overall performance improvement on this GPU was not as dramatic as the GT 755m GPU. This is likely because the GTX 1070 is more powerful. Compared to the GT 755m, the GTX 1070 has 5x more cores, 3x increased memory bandwidth and 8x more L2 cache. As a result, the effects of a poor access pattern, for example, is mitigated by the faster bandwidth and increased cache size.

7.1. ACO Convergence

We also evaluated the efficacy of our extensions to the ACO algorithm. Figure 14 shows the performance of the best-so-far solution for each evaluation of an ant throughout the ACO algorithm. Three configurations of the CA framework are compared on a subset of the benchmarks. CA base is the ACO algorithm without any of the extensions described earlier. Ants make decisions based only on the pheromone coefficient. CA with metrics shows the performance of the ACO algorithm when performance metrics are included as the *a priori* parameter to equation 2. Finally, CA full shows the performance when metrics and the regression tree are included.

Results suggest that the use of a regression tree allows the CA full configuration to break out of local minimums that trap the other two configurations. For most benchmarks, the use of metrics led to a modest improvement over CA base. There are some exceptions however. In figure 14(b), CA with metrics performs slightly worse than CA base. For the gesummv benchmarks on both GPUs, CA with metrics performs similarly to CA full. This suggests there is some benefit to including performance metrics into the ACO algorithm.

7.2. Comparison

To provide some context for our results, we compared them with those achieved in another recent work, that of Shirako et al. [48] who developed a framework called *PolyAST*. PolyAST achieves higher performance through integrating separate schedules for block-level and thread-level parallelism by ways of *superposition*. They demonstrated their framework using the Poly-Bench suite in conjunction with two GPU architectures: (1) an older high-end

Coding Ants	Coding Ants	PolyAST	PolyAST
GeForce GTX 1070	GeForce GT 755m	Tesla M2050	Tesla K80
1.3	1.5	0.9	1.2

Figure 11: Median speed-ups for the PolyBench benchmarks *2mm, 3mm, covariance, doitgen, gemm, gemver, gsummv, jacobi-2d* and *mvt* for the Coding Ants and PolyAST frameworks using the NVIDIA boards listed in the table header.

GPU – the NVIDIA Tesla M2015 and (2) a more recent one – the NVIDIA Tesla K80. While these are GPU boards rather different from the ones we tested our framework on, we can neutralize these differences by comparing just the speed-ups obtained over the default PPCG generated code. For our comparison we took the maximum of the O1, O2, O3 speed-ups for Coding Ants. For PolyAST we divided the two speed-up numbers they reported, that is, the speed-up of PPCG over a gcc-O3 compilation (resulting in sequential code) and the speed-up of PolyAST over that gcc-O3 compilation.

There were nine PolyBench benchmarks that both Shirako et al. and we tested. Figure 15 visualizes eight of these nine benchmarks.The *doitgen* benchmark scored a speed-up of over 200 with both Tesla boards for PolyAST and we omitted it because it would dominate the plot. Further, we also calculated the median speedup obtained for each GPU model and algorithm, shown in the table of Figure 11.

Given these analyses we observe a slight advantage (around 10%) of Coding Ants over PolyAST in the median case. But there are some benchmarks in which CA shines and some in which PolyAST does better. Others show similar performance gains. Recalling that we have also observed this diversity for CA's different optimization levels, it appears that the best optimization strategy depends on the input code (here, the benchmark); there are no clear winners overall. In practice, one would need to try out the different strategies and see which performs best for a specific piece of code.

8. Conclusions

In this work we presented the Coding Ants framework, a tool for using ACO for GPU based auto-tuning. We showed how GPU optimization

* Timings do not include CUDA memcpy as it accounts for $\geq 50\%$ of execution time.

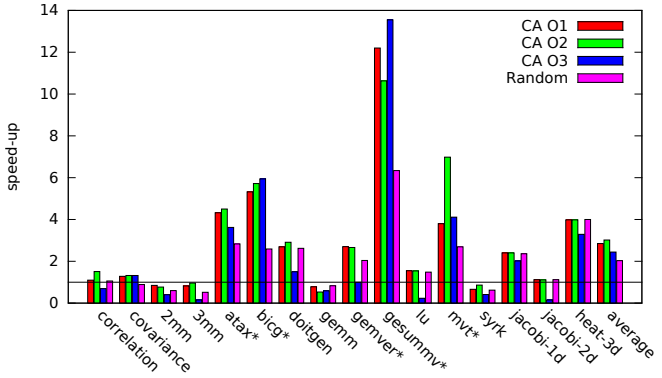


Figure 12: Speed-up over the default PPCG generated code on the NVIDIA GeForce GT 755m GPU using three optimization levels: O1, O2, O3, and a random search.

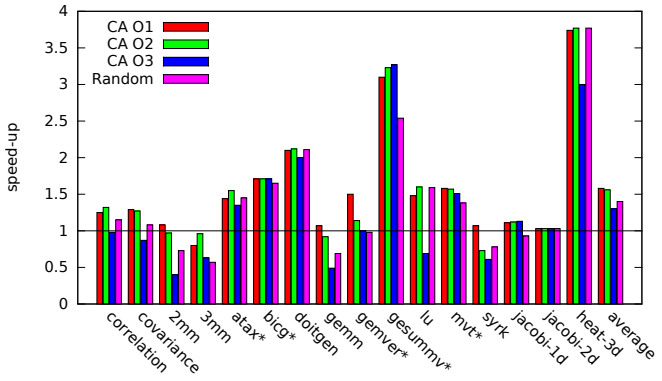


Figure 13: Speed-up over the default PPCG generated code on the NVIDIA GeForce GTX 1070 GPU using three optimization levels: O1, O2, O3, and a random search.

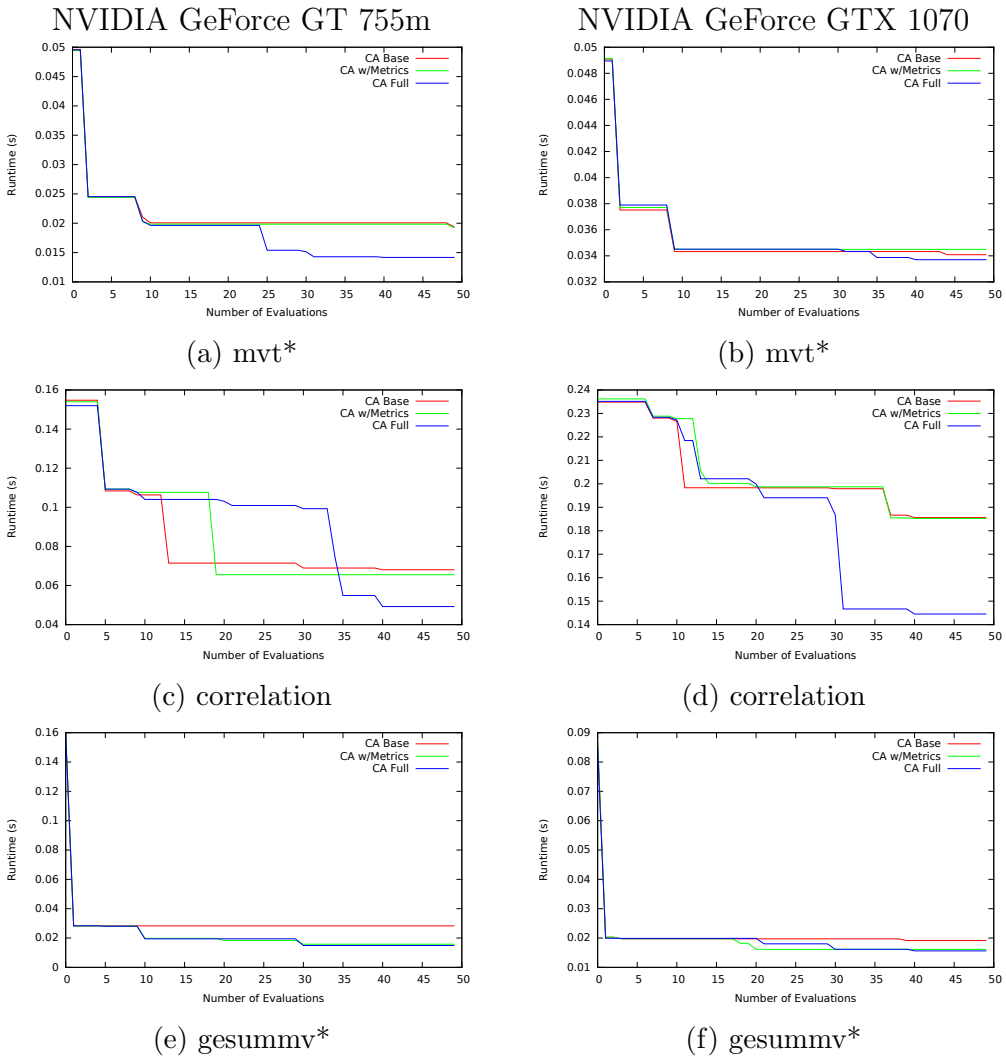


Figure 14: Evolution of the best-so-far solution for the base CA framework, CA with performance metrics, and CA using metrics and a regression tree. Results from three benchmarks are shown on the NVIDIA GeForce GT 755m (a)(c)(e), and the NVDIA GeForce GTX 1070 (b)(d)(f).

can be represented through a, possibly cyclic, optimization graph. We also presented several optimizations that are not typically considered with GPU auto-tuners. Experimental evaluation of three levels of optimization that give increasing control to the CA framework was also performed. Results show that the O1 and O2 optimization levels are the most promising, while

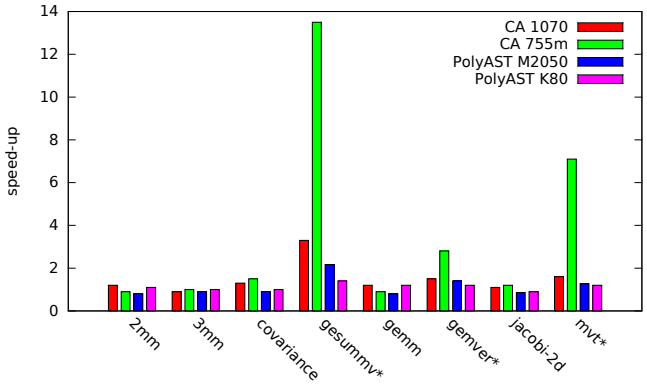


Figure 15: Speed-up over default PPCG generated code for Coding Ants and PolyAST, each running on different GPU hardware.

the O3 optimization performed the worst. In general, there are too many ways to harm performance by selecting random schedules. Use of a cost model to select the schedule analytically is shown to be the best choice for most benchmarks. Finally, we presented several extensions to the ACO algorithm to make it more amenable to performance optimization. Experiments showed using performance metrics and regression trees in particular led to better quality solutions over ACO alone.

Acknowledgements

This work was supported by the National Science Foundation grants CNS-0435060, CCR-0325197, EN-CS-0329609, and IIS-1527200, as well as the MSIP (Ministry of Science, ICT and Future Planning), Korea, under the "ITCCP Program" directed by NIPA

References

[1] (2011). The openacc application programming interface, .

[2] (2012). *NVIDIA CUDA C Programming Guide 4.2*. Technical Report.

[3] (2014). *Visual Profiler User's Guide*. Technical Report.

[4] Agakov, F., Bonilla, E., Cavazos, J., Franke, B., & Fursin, G. (2006). Using machine learning to focus iterative optimization. In *Proceedings*

of the International Symposium on Code Generation and Optimization (pp. 295–305).

[5] AlZayer, F. R. (2015). *ACCTuner: OpenACC Auto-Tuner For Accelerated Scientific Applications*. Master's thesis King Abdullah University of Science and Technology.

[6] Ansel, J., Kamil, S., Veeramachaneni, K., Ragan-Kelley, J., Bosboom, J., O'Reilly, U.-M., & Amarasinghe, S. (2014). Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd international conference on Parallel architectures and compilation* (pp. 303–316). ACM.

[7] Ashouri, A. H., Mariani, G., Palermo, G., Park, E., Cavazos, J., & Silvano, C. (2016). Cobayn: Compiler autotuning framework using bayesian networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, *13*.

[8] Benabderrahmane, M.-W., Pouchet, L.-N., Cohen, A., & Bastoul, C. (2010). The polyhedral model is more widely applicable than you think. In *Proceedings of the International Conference on Compiler Construction* CC '15. Paphos, Cyprus.

[9] Blum, C. (2005). Ant colony optimization: Introduction and recent trends. *Physics of Life Reviews*, *2*, 353–373.

[10] Blum, C., & Dorigo, M. (2004). The hyper-cube framework for ant colony optimization. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, *34*, 1161–1172.

[11] Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., & Rohou, E. (1998). Iterative compilation in a non-linear optimisation space. *Workshop on Profile and Feedback-Directed Compilation*, .

[12] Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., & Sadayappan, P. (2008). Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*.

[13] Breiman, L., Friedman, J., Stone, C. J., & Olshen, R. (1984). *Classification and Regression Trees*. Taylor and Francis.

[14] Bullnheimer, B., Hartl, R. F., & Strauss, C. (1999). A new rank-based version of the ant system: A computational study. *Central European J Operations Res Econom*, *7*, 25–38.

[15] Chaimov, N., Biersdorff, S., & Malony, A. D. (2013). Tools for machine-learning-based empirical autotuning and specialization. *International Journal of High Performance Computing Applications*, *27*, 403–411.

[16] Chen, C. (2007). *Model-Guided Empirical Optimization for Memory Hierarchy*. Ph.D. thesis University of Southern California.

[17] Christen, M., Schenk, O., & Burkhart, H. (2011). Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *Proceedings of the 2011 IEEE 25th International Parallel & Distributed Processing Symposium* IPDPS '11 (pp. 676–687). Washington DC, USA.

[18] Dorigo, M. (1992). *Optimization, learning and natural algorithms*. Ph.D. thesis Politecnico di Milano, Italy Milan, Italy.

[19] Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics – Part B*, *26*, 29–41.

[20] Enterprise, C. (2010). *Hmpp:hybrid compiler for many core applications workbench user guide.*. Technical Report.

[21] Feautrier, P. (1991). Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, *20*, 23–53.

[22] Feautrier, P. (1992). Some efficient solutions to the affine scheduling problem: part i. one-dimensional time. *International Journal of Parallel Programming*, *21*, 313–348.

[23] Feautrier, P. (1992). Some efficient solutions to the affine scheduling problem: part ii. multidimensional time. *International Journal of Parallel Programming*, *21*, 389–420.

[24] Grauer-Gray, S., Xu, L., Searles, R., Ayalasomayajula, S., & Cavazos, J. (2012). Auto-tuning a high-level language targeted to gpu codes. *Innovative Parallel Computing*, .

[25] Harris, M. (2007). Optimizing parallel reduction in cuda. Presentation packaged with CUDA Toolkit.

[26] Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*, 671–680.

[27] Kisuki, T., Knijnenburg, P. M. W., O'Boyle, M. F. P., Bodin, F., & Wijshoff, H. A. G. (1999). A feasibility study in iterative compilation. *High Performance Computing*, (pp. 121–132).

[28] Kolesnichenko, A., Poskitt, C. M., & Nanz, S. (2017). Safegpu: Contract-and library-based gpgpu for object-oriented languages. *Computer Languages, Systems & Structures*, *48*, 68–88.

[29] Liao, C., Quinlan, D. J., Vuduc, R., & Panas, T. (2009). Effective source-to-source outlining to support whole program empirical optimization. In *International Workshop on Languages and Compilers for Parallel Computing* LCPC '09. Newark Delaware.

[30] Lim, R. V., Norris, B., & Malony, A. D. (2017). Auto-tuning gpu kernels via static and predictive analysis, . URL: `https://arxiv.org/abs/1701.08547`.

[31] Luitjens, J. (2014). Faster parallel reductions on kepler. URL: `https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/` presentation packaged with CUDA Toolkit.

[32] Magni, A., Grewe, D., & Johnson, N. (2013). Input-aware auto-tuning for directive based gpu programming. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units* GPGPU-6 (pp. 66–75). New York, NY.

[33] Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *Computer Journal*, *7*, 308–313.

[34] Nystrom, N., White, D., & Das, K. (2011). Firepile: run-time compilation for gpus in scala. In *ACM SIGPLAN Notices* (pp. 107–116). ACM volume 47.

[35] OpenMP (1997). Openmp – api specification for parallel programming. Http://openmp.org.

[36] Papenhausen, E., & Mueller, K. (2013). "rapid rabbit: Highly optimized gpu accelerated cone-beam ct reconstruction. In *Nuclear Science Symposium and Medical Imaging Conference*.

[37] Papenhausen, E., Zheng, Z., & Mueller, K. (2013). Creating optimal code for gpu-accelerated ct reconstruction using ant colony optimization. *Medical Physics*, *40*.

[38] Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., & Vasilache, N. (2011). Loop transformations: convexity, pruning and optimization. *ACM SIGPLAN Notices*, *46*, 549–562.

[39] Puschel, M., Moura, J. M. F., Johnson, J. R., Padua, D., Veloso, M. M., Singer, B. W., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W., & Rizzolo, N. (2005). Spiral: Code generation for dsp transforms. In *Proceedings of the IEEE, special issue on Program Generation, Optimization and Platform Adaptation* (pp. 232–275). volume 93.

[40] Puschel, M., Singer, B., Xiong, J., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., & Johnson, R. W. (2004). Spiral: A generator for platform-adapted libraries of signal processing algorithms. *International Journal of High Performance Computing Applications*, *18*, 21–45.

[41] Qasem, A., Kennedy, K., & Mellor-Crummely, J. (2006). Automatic tuning of whole applications using direct search and a performance-based transformation system. *The Journal of Supercomputing*, *36*, 183–196.

[42] Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, *1*, 81–106.

[43] Rauchwerger, L., Amato, N. M., & Padua, D. A. (1995). Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th ACM International Conference on Supercomputing*. Barcelona, Spain.

[44] Rudy, G., Khan, M. M., Hall, M., Chen, C., & Chame, J. (2010). A programming language interface to describe transformations and code

generation. In *International Workshop on Languages and Compilers for Parallel Computing* LCPC '10. Berlin, Heidelberg.

[45] Russel, S. J., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. (2nd ed.). Pearson Education.

[46] Sastry, K., Goldberg, D., & Kendall, G. (2005). Search methodologies. name of chapter: Genetic Algorithms. (pp. 97–125). Springer.

[47] Schrijver, A. (1998). *Theory of Linear and Integer Programming*. Wiley.

[48] Shirako, J., Hayashi, A., & Sarkar, V. (2017). Optimized two-level parallelization for gpu accelerators using the polyhedral model. In *Proceedings of the 26th International Conference on Compiler Construction* (pp. 22–33). ACM.

[49] Steuwer, M., Remmelg, T., & Dubach, C. (2017). Lift: a functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (pp. 74–85). IEEE Press.

[50] Stutzle, T., & Hoos, H. H. (2000). Max–min ant system. *Future Generation Computing Systems*, *16*, 889–914.

[51] Tabatabaee, V., Tiwari, A., & Hollingsworth, J. K. (2005). Parallel parameter tuning for applications with performance variability. In *Proceedings of the 2005 ACM/IEEE conference on supercomputing* SC '05. Washington DC, USA.

[52] Tapus, C., Chung, I.-H., & Hollingsworth, J. K. (2002). Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on supercomputing* SC '02. Baltimore, Maryland.

[53] Thies, W., Karczmarek, M., & Amarasinghe, S. (2002). Streamit: A language for streaming applications. In *International Conference on Compiler Construction* (pp. 179–196). Springer.

[54] Tillmann, M., Karcher, T., Dachsbacher, C., & Tichy, W. F. (2014). Application-independent autotuning for gpus. *Parallel Computing: Accelerating Computational Science and Engineering*, *25*, 626–635.

[55] Tiwari, A., Chen, C., Chame, J., Hall, M., & Hollingsworth, J. (2009). A scalable autotuning framework for compiler optimization. In *Proceedings of the 2009 International Symposium on Parallel & Distributed Processing* IPDPS '09. Rome, Italy.

[56] Tiwari, A., Hollingsworth, J. K., Chen, C., Hall, M., Liao, C., Quinlan, D. J., & Chame, J. (2011). Auto-tuning full applications: A case study. *International Journal of High Performance Computing Applications*, *25*, 286–294.

[57] Verdoolaege, S., Juega, J. C., Cohen, A., Gomez, J. I., Tenllado, C., & Catthoor, F. (2013). Polyhedral parallel code generator for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, *9*.

ALGORITHM 2: ACO + Pluto Algorithm [12]

Input: Generalized dependence graph $GDG = (V, E)$ (includes dependence polyhedra $\mathcal{P}_e, e \in E$)

S_{max}: statement with maximum domain dimensionality
for *each dependence* $e \in E$ **do**
    Build legality constraints: apply Farkas Lemma on $\phi(\mathbf{t}) - \phi(\mathbf{s}) \geq 0$, and eliminate all Farkas multipliers
    Build communication volume/reuse distance bounding constraints
    Aggregate constraints from both into $C_e(i)$
end
repeat
    $C = \emptyset$
    $D = \emptyset$
    for *each dependence edge* $e \in E$ **do**
        $C \rightarrow C \cup C_e(i)$
        $D \rightarrow D \cup \mathcal{P}_e$
    end
    repeat
        if *O3* **then**
            Compute the set of valid scheduling hyperplanes such that $\phi \supseteq D$ and aggregate them into V
            Use ACO to select a schedule: $ConstructSolution(V)$
        else
            Compute lexicographic minimal solution to C
        end
        if *Solution exists* **then**
            Add orthogonality constraint
            if *O2 or O3* **then**
                Use ACO to select which, if any, dependences to cut: $ConstructSolution(GDG)$
                if *Cut Dependences* **then**
                    Insert the appropriate *splitter* in the transformation matrices of the statements
                    continue
                end
            end
            Use ACO to select whether to perform parallel reduction: $ConstructSolution(GDG)$
        end
    until *no solution is found or* $H^{\perp}_{S_{max}} = \boldsymbol{0}$;
    Compute E_c: carried dependences
    $E \rightarrow E - E_c$: update $GDG(V, E)$
until $H^{\perp}_{S_{max}} = \boldsymbol{0}$ *and* $E = \emptyset$;

Output: A transformation matrix for each statement

Table 2: Performance Metrics

Metric	Description
Executed IPC	Instructions executed per cycle
Achieved Occupancy	Ratio of active warps to the maximum number of warps per cycle
Global Store Throughput	Global memory store throughput
Global Load Throughput	Global memory load throughput
Cache Hit Rate	Average hit rate for L2 and texture cache