Hindawi
Scientific Programming
Volume 2018, Article ID 6093054, 24 pages
https://doi.org/10.1155/2018/6093054

Research Article

A Strategy for Automatic Performance Tuning of Stencil Computations on GPUs

Joseph D. Garvey and Tarek S. Abdelrahman

Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada M5S 3G4

Correspondence should be addressed to Tarek S. Abdelrahman; tsa@eecg.toronto.edu

Received 20 September 2017; Revised 11 March 2018; Accepted 17 April 2018; Published 28 May 2018

Academic Editor: Basilio B. Fraguela

We propose and evaluate a novel strategy for tuning the performance of a class of stencil computations on Graphics Processing Units. The strategy uses a machine learning model to predict the optimal way to load data from memory followed by a heuristic that divides other optimizations into groups and exhaustively explores one group at a time. We use a set of 104 synthetic OpenCL stencil benchmarks that are representative of many real stencil computations. We first demonstrate the need for auto-tuning by showing that the optimization space is sufficiently complex that simple approaches to determining a high-performing configuration fail. We then demonstrate the effectiveness of our approach on NVIDIA and AMD GPUs. Relative to a random sampling of the space, we find configurations that are 12%/32% faster on the NVIDIA/AMD platform in 71% and 4% less time, respectively. Relative to an expert search, we achieve 5% and 9% better performance on the two platforms in 89% and 76% less time. We also evaluate our strategy for different stencil computational intensities, varying array sizes and shapes, and in combination with expert search.

1. Introduction

Stencil computations appear in many domains including image processing [1], partial differential equation solvers [2], and cellular automata [3]. They are often part of applications that demand high performance and they have abundant parallelism. Thus, they are excellent candidates for acceleration on Graphics Processing Units (GPUs).

However, programmers must apply *optimizations* to their stencil code in order to exploit the underlying GPU architecture. They must ensure memory coalescing, use local, texture, or read-only memories, reduce thread divergence, and select kernel launch configurations that balance parallelism with resource usage [4]. The impact of each optimization is often difficult to assess, particularly when combined with other optimizations. Programmers are left to explore a large space of optimization *configurations*, i.e., combinations of optimizations and their parameters, to find good performing ones. This exploration entails running the code for each configuration of interest—a process that can take months of compute time [5].

Thus, there has been considerable interest in performance *auto-tuning*, i.e., automatically exploring the space of configurations in efficient ways to determine a good combination of optimizations to apply. A common approach is to use expert knowledge of the stencil computation and the underlying GPU architecture to limit the space, then to exhaustively search through it; we refer to this strategy as *expert search*. This approach requires new expert knowledge for every stencil computation and for every new GPU architecture. In addition, it can still require exploring a large number of configurations.

In this work, we present an auto-tuning strategy for searching through the space of possible optimization configurations for good performing ones. Our approach tackles the large space in two ways. First, it partitions the space based on memory optimization and uses machine learning to predict the partition containing the best configuration. Second, it divides the remaining optimizations into groups and tunes the groups independently. This approach explores fewer configurations than expert search approaches while obtaining as good or better performing configurations.

We first define a set of optimizations that are commonly applied to stencil computations. We train a random forest [6] machine learning model to predict the optimal memory type based on static program features. We then develop three alternative heuristics for grouping the remaining optimizations. The first groups the optimizations along the dimensions of the grid in which the GPU threads are organized. The second puts each optimization, applied to all dimensions of the grid, in a group by itself. The third is a hybrid approach that combines features of the first two.

We evaluate our strategy on an NVIDIA GTX Titan and an AMD Radeon R390 using 104 synthetically generated OpenCL stencil kernels with wide ranges of stencil patterns (dense, star, no-corner, diamond, and thumbtack) and radii (0 to 5). In comparison to an expert search approach, our strategy with the hybrid approach takes 89% and 76% less tuning time on the NVIDIA and AMD platforms, respectively, while finding a best configuration that is on average 5% and 9% better, respectively. Compared to a random sampling of the space, our strategy with the same heuristic takes 71% and 4% less time on the NVIDIA and AMD platforms, respectively, while finding a final configuration that is on average 12% and 32% better, respectively.

The rest of this paper is structured as follows. Section 2 provides background material. Section 3 defines stencil computations. Section 4 describes the optimizations we tune. Section 5 demonstrates the interestingness of the configurations space. Section 6 details our auto-tuning strategy. Section 7 presents the infrastructure we have developed to evaluate that strategy. Section 8 reports our evaluation. Section 9 outlines related work and Section 10 concludes.

2. Background

2.1. GPU Architecture. A GPU consists of clusters of simple cores. These clusters are known as *streaming multiprocessors* (SMXs) on the NVIDIA GeForce GTX Titan [7] and as *compute units* (CUs) on the AMD Radeon R9 390 [8]. For simplicity, we will use the term SMX to refer to them in the rest of this document. SMXs do not possess sufficient instruction issuing ability to keep all of their cores busy executing different instructions and so, to keep the cores fully utilized, threads are launched in groups, known as warps on NVIDIA and wavefronts on AMD, that execute in lockstep, performing the same operation on different data. To achieve good performance on these platforms, it is essential that applications expose sufficient parallelism to allow a high utilization of the many cores.

GPUs have a multi-tiered memory hierarchy whose characteristics have important performance implications. One of the most important factors affecting performance is how data is loaded from *global memory*, the slow, off-chip memory of the GPU. If multiple consecutive threads all access a contiguous, aligned chunk of memory of a particular size, these accesses are *coalesced* into a single memory transaction [4, 8]. Coalesced accesses significantly reduce memory access time by reducing the number of accesses.

GPUs also possess a software-managed cache that is shared at the SMX level, known as shared memory on NVIDIA [7] or local data share (LDS) on AMD [8]. Use of this memory can remove duplicated reads when threads running on the same SMX access the same data.

Many GPUs also possess *texture memory*, a hardware managed cache with multidimensional locality that is used to store image data types [4]. In modern architectures, this cache is being phased out in favour of more generic *read-only* data caches that fulfill a similar purpose [7, 8].

2.2. OpenCL Programming Model. We use Open Computing Language (OpenCL) [9], an industry standard language for programming heterogeneous parallel systems. Its paradigm consists of a *host* from which programs, known as *kernels*, are *launched* and one or more *devices* on which those kernels execute. Many instances of the kernel code are executed concurrently, each by a *work-item*, i.e., a thread. Work-items exist in a multidimensional grid and are identified by their indices, known as *global IDs*, in each dimension of this grid.

Each work-item is part of a *work-group*, a multidimensional collection of work-items that execute on the device at the same time. The IDs of a work-item in each dimension within its work-group are known as its *local IDs*. Work-items in the same work-group are mapped to the same SMX, making it possible to synchronize across them. Work-items in the same work-group share a software-managed cache known as *local memory* which is realized through shared memory/LDS.

Various resource requirements of the kernel can impose limitations on the maximum work-group size. For example, each SMX has a limited number of registers as well as a limited amount of shared memory. Thus the register usage or local memory usage of the kernel could be the limiting factor that determines the maximum work-group size.

The sizes of the two grids mentioned, global and local, are provided by the host program at kernel launch time and together constitute the *launch configuration* of the kernel. OpenCL separates the compilation of a kernel from the launching of that kernel and as a result the total number of work-items that will be launched and the size of each work-group are not normally known at kernel compile time.

OpenCL defines a number of image data types that can be used to store array data. Often, these images can be cached in the read-only caches.

Finally, OpenCL also supports vector types as well as arithmetic on those vector types. In the context of GPUs, these vector operations can be mapped to specialized vector hardware or distributed amongst cores. While one expects these vector operations to improve arithmetic performance, it is less obvious that even reads and writes to global memory can be accelerated by acting on vectorized data [10].

3. Stencil Computations

A stencil computation produces one or more output arrays as a function of one or more input arrays in such a way that each output element is a function of some input elements at fixed offsets relative to the index of the output element. This fixed set of offsets defines the *stencil*. Algorithm 1 shows sequential

```
void stencil_computation (
    float (∗array1) [Y_SIZE] [X_SIZE],
    float (∗array2) [Y_SIZE] [X_SIZE])
{
    float (∗in) [Y_SIZE] [X_SIZE] = array1;
    float (∗out) [Y_SIZE] [X_SIZE] = array2;
    for (int t = 0; t < T_MAX; ++t) {
        for (int z = 0; z < Z_SIZE; ++z)
            for (int y = 0; y < Y_SIZE; ++y)
                for (int x = 0; x < X_SIZE; ++x) {
                    float temp0 = in [z+C_0z] [y+C_0y] [x+C_0x];
                    float temp1 = in [z+C_1z] [y+C_1y] [x+C_1x];
                    ⋮
                    float tempN = in [z+C_Nz] [y+C_Ny] [x+C_Nx];
                    out [z] [y] [x]= f(temp0, temp1, . . . , tempN);
                }
        // Swap in and out pointers
    }
}
```

Algorithm 1: Generic 3D stencil computation.

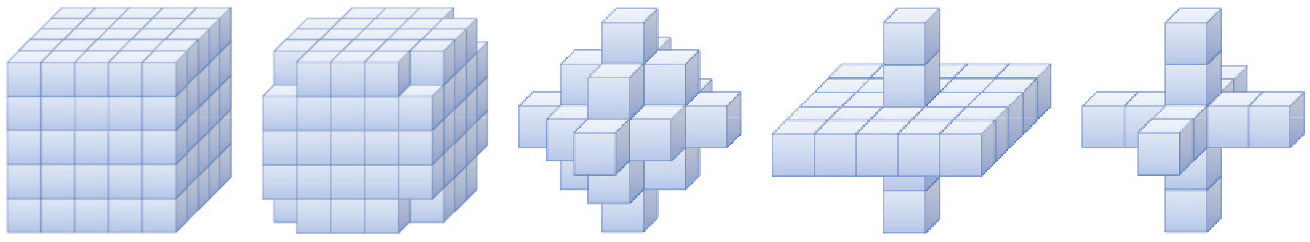


Figure 1: Stencil patterns.

C code for the general form of a stencil computation with three-dimensional input and output arrays. The loops `x`, `y`, and `z` are known as the *spatial* loops and they sweep through the elements of the output array, `out`. They compute each of its elements as a function, `f`, of the stencil defined by the set of constants C_{ij}, the ith offset in the jth dimension. `X_SIZE` and `Y_SIZE` are constants whose definitions are omitted for brevity. The `t` loop repeats this process and is known as the *time* loop.

It is possible for the input and output arrays of a stencil computation to be the same, e.g., in successive overrelaxation [11]. This introduces loop-carried data dependencies that render such stencils less suited for GPUs. Thus, we restrict ourselves to stencil computations in which the input and output arrays are different. For simplicity, we only consider stencil programs with one input and one output array of the same dimensionality.

Stencil computations can have a variety of different stencil patterns, as shown in Figure 1. The stencil *radius* is the maximum distance between any element of the stencil and the centre of the stencil. The examples in Figure 1 all have a stencil radius of two. The stencil *size* is the number of input elements used to compute each output element. The stencil *density* is equal to the size of the stencil divided by the size of the smallest rectangular prism that bounds the entire stencil.

When implemented naively in OpenCL, the spatial loops of a stencil become a kernel, as shown in Algorithm 2. The computation of each output element is mapped to a single work-item and the `for` loops are replaced with OpenCL API calls to `get_global_id()`. Thus, in this example, a three-dimensional global grid is used. We use the letters x, y, and z to refer to the 0th, 1st, and 2nd dimensions of this grid, respectively. The time loop is executed by the host, causing it to repeatedly launch the kernel. We focus on optimizing this kernel and auto-tune a single invocation of it. Thus, we do not consider optimizing the time loop (e.g., by tiling as done by Grosser and others [12, 13]).

4. Optimizations

4.1. Work-Group Size. The work-group size is the number of work-items in a work-group in each dimension. This size can have a significant impact on performance because it affects available parallelism. It needs to be large enough to utilize the hardware, but not so large as to reduce parallelism by reducing the number of work-groups that can execute at once.

```
__kernel void stencil_computation (
    global float (*in) [Y_SIZE] [X_SIZE],
    global float (*out) [Y_SIZE] [X_SIZE])
{
    int x = get_global_id (0);
    int y = get_global_id (1);
    int z = get_global_id (2);
    float temp0 = in [z+C_{0z}] [y+C_{0y}] [x+C_{0x}];
    float temp1 = in [z+C_{1z}] [y+C_{1y}] [x+C_{1x}];
    ⋮
    float tempN = in [z+C_{Nz}] [y+C_{Ny}] [x+C_{Nx}];
    out [z] [y] [x]= f(temp0, temp1, . . . , tempN);
}
```

Algorithm 2: Generic 3D OpenCL stencil kernel.

A large work-group size can also exhaust other resources such as registers or local memory, reducing performance or rendering a configuration unexecutable.

4.2. Block Merging. Block merging combines the work done by adjacent work-items into a single work-item. Thus, it is a form of tiling. The number of work-items merged together is referred to as the *block merge factor*. A loop is introduced in the kernel code to compute multiple output elements instead of only one.

Block merging increases thread work granularity, thereby mitigating some of the overhead of launching threads. It also allows duplicate reads to be removed if the introduced loop is unrolled. Specifically, it allows global memory accesses to be replaced by register reads if the merged work-items require the same input element in their computation. The removal of these duplicate reads requires a compiler that is able to unroll the loop created by merging identify duplicate reads and remove them.

However, block merging increases the register use of each work-item. This in turn can decrease the maximum work-group size and the number of concurrent work-groups executing on an SMX. Further, block merging in the innermost dimension of the global grid can uncoalesce already coalesced accesses. These factors negatively impact performance and make it hard to predict the benefit of block merging.

4.3. Cyclic Merging. Cyclic merging combines nonadjacent work-items such that work is assigned to work-items in a work-group in a round-robin fashion. The number of work-items merged in this manner is referred to as the *cyclic merge factor*. Cyclic merging introduces a loop into the kernel code for each dimension merged and also requires some changes to the array index calculations.

Cyclic merging increases thread work granularity and has the benefit that when applied to the innermost dimension of the grid it does not disrupt existing memory coalescing. However, it is unlikely to remove duplicate reads since the elements accessed by the merged work-items are likely far apart. Like block merging, it can limit parallelism by increasing register usage and decreasing the maximum work-group size and the maximum number of concurrent work-groups. As a result, it does not always improve performance.

4.4. Local Memory Caching. This optimization refers to caching the input data into local memory before reading it. This caching is done cooperatively by the threads in a way that ensures memory coalescing when possible. The amount of data cached in local memory is the minimum needed by the stencil computations, thus reducing the total amount of local memory needed. Reads from global to local memory are introduced at the beginning of the kernel, the existing reads from global memory are replaced with reads to local memory, and barriers are added between the two types of reads for synchronization.

Local memory can improve kernel performance when there are repeated accesses to the same data within a work-group. It can also improve memory coalescing when memory transactions are otherwise uncoalesced. However, large local memory requirements can unnecessarily restrict the maximum work-group size, limiting available parallelism. Furthermore, use of local memory adds the overhead of extra memory accesses and synchronization. Thus, it is not always beneficial [14].

4.5. Vectorization. This optimization converts reads from the input array, arithmetic operations on intermediate results, and writes to the output array into vector operations that exploit specialized vector hardware. It only applies to kernels that are block merged in the x dimension and only to those operations that were formerly done by multiple work-items that, after merging, are accomplished by a single work-item. The two optimizations in conjunction replace parallelism via work-items with parallelism via vectorization.

Reads that align to vector boundaries only require changing the data types of relevant variables to vector types. Input reads that do not align to a vector boundary require two vector reads and the composition of an interim vector result. Output writes are always aligned to vector boundaries.

4.6. Image Memory. This optimization stores the input array as a read-only OpenCL image data type rather than a standard OpenCL memory buffer. This allows the kernel to exploit texture or read-only caches on the GPU. It requires changing the data type of the input array and its allocation on the host. Further, OpenCL API calls must be used to access the elements rather than normal array subscripts. The texture/read-only cache is hardware managed and thus requires no additional changes to the kernel.

The use of texture/read-only caches is possible since the input array remains constant for the duration of the kernel. The swapping of the input and output arrays is performed in the outer time loop (see Section 3) outside the kernel between its successive invocations. To ensure that this swapping incurs no additional overhead, the target platform must support a specific extension, namely, the `cl_khr_image2d_from_buffer` extension. The host code should allocate the input and output arrays first as buffers and

Table 1: Optimization abbreviations and allowable values. N denotes the input/output array size in each dimension. X, Y, and Z correspond to the x, y, and z dimensions of the work-item grid, respectively.

Optimization	Abbreviation	Range of values	
		Minimum	Maximum
Work-group size	WX, WY, WZ	1	NX, NY, NZ
Block merge factor	BX, BY, BZ	1	NX, NY, NZ
Cyclic merge factor	CX, CY, CZ	1	NX, NY, NZ
Vector width	VX	1	16
Local memory	N/A	0 (no)	1 (yes)
Image memory	N/A	0 (no)	1 (yes)

should then create 2D images from those buffers. This does not create a copy but instead allows the data to be accessed as both a buffer and an image. Thus, in each stencil iteration, the host passes the input argument as an image and the output as a buffer.

This optimization can improve performance by replacing expensive global memory accesses with accesses to the texture/read-only cache. However, local memory can accomplish the same and using both simultaneously adds overhead for no benefit. Determining which to use for a given kernel is nontrivial.

4.7. Optimization Space. All of the optimizations are summarized in Table 1 along with abbreviations used throughout the paper to refer to them. Each optimization takes on a *value* from the given range. Thus, an *optimization configuration* is the full set of values, one for each optimization. For simplicity, we restrict optimization values to powers of 2. Further, the possible values for many of the optimizations depend on the other optimization values. The product of W, B, and C for a given dimension (e.g., WX or BY) must be less than or equal to N in that dimension. In addition, VX must be less than or equal to BX. These restrictions result in a total optimization space of over 50 million configurations. Many of these configurations are not actually executable because they violate some limitation of the hardware; e.g., they use more local memory than is available on the device. Nonetheless, the space is prohibitively large to explore exhaustively.

In order to make this space more manageable, some obviously bad configurations are never considered: (1) ones in which the block merging factor in the x dimension is greater than the vectorization factor, since this reduces memory coalescing and experience has shown that this always results in poor performance, and (2) ones with block merging in the y and z dimensions because current GPU compilers are not able to remove the duplicate reads produced by block merging (see Section 4.2). (Nonetheless, block merging can have a significant impact on performance, should OpenCL compilers support the removal of duplicate reads. For example, in a radius 2 1D (oriented in the y dimension) kernel, 4 out of 10 loads are duplicates. Manually removing these duplicates speeds up the execution of the kernel, measured on the NVIDIA platform, by a commensurate 41%.) Further, we opt to allow only one of four forms of data loading: global memory without vectorization, global memory with vectorization, local memory, or image memory. This is done because vectorization, local memory, and image memory all effectively target the same thing: the way in which data is loaded from global memory. This choice will be referred to as the *data loading technique*. While there could be some benefit to considering vectorization in conjunction with the other optimizations from a purely computational perspective, we suspect that this benefit is small and thus not worth the increase in the optimization space. These restrictions reduce the optimization space to 475,875 possible configurations. Although this is a huge reduction in the number of configurations, it would still be impractical to examine this reduced space exhaustively. Compiling and running these configurations for all the kernels would take over a year.

5. Interestingness

We demonstrate the need for auto-tuning by showing that the optimization space is "interesting". That is, it is sufficiently complex that simple approaches to determining a high-performing configuration fail. To this end, we show the following:

(1) The majority of configurations in the space perform poorly.

(2) The optimal value for each optimization varies from kernel to kernel.

(3) The optimizations interact, requiring that they be explored together and not individually.

(4) The best optimization configuration for one kernel does not necessarily perform well for other kernels.

5.1. Random Sampling. Ideally, the entire optimization space would be examined to demonstrate the above four points. However, this is not feasible in a reasonable amount of time for a single kernel, let alone for a large number of them. Consequently, the exploration of the optimization space is performed by randomly sampling configurations. These configurations are used as a proxy for the full optimization space.

When choosing the sample size, it is important to ensure that enough of the space is searched to find a good performing configuration without taking an inordinate amount of time.

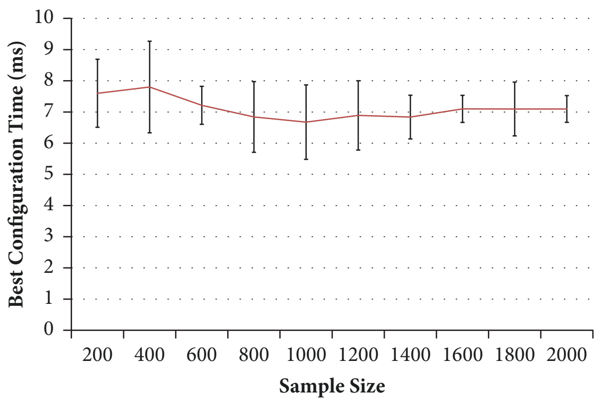


FIGURE 2: Best runtime versus sample size.

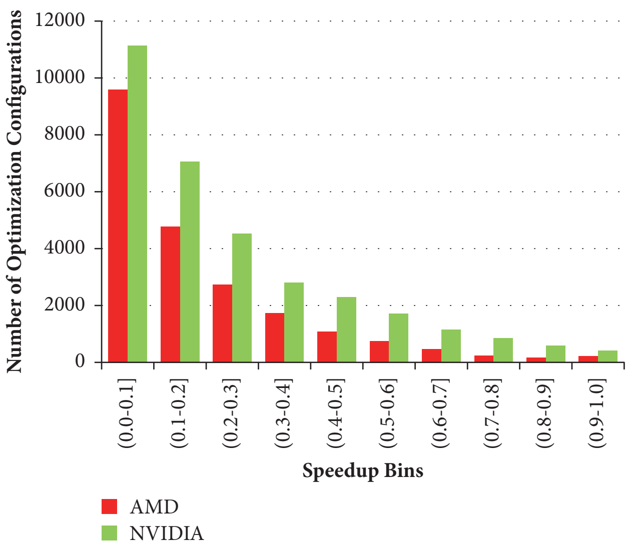


FIGURE 3: Distribution of configuration speedups.

In order to determine the appropriate sample size, we ran samples of various sizes on the NVIDIA GTX Titan for five of the 104 kernels (one from each of the stencil types mentioned in Section 3). The five kernels chosen all had a dimensionality of three and a radius of two as these parameters result in a significant amount of variability across the five kernels while running in a reasonable amount of time. For each of the selected kernels, samples were taken ranging in size from 200 to 2000 optimization configurations in increments of 200 and the best runtime of each sample was determined. To make the results more statistically sound, each size was randomly sampled 10 times and an average (mean) and variance of the best runtime was determined. The results of this process for the dense kernel are shown in Figure 2. The results for the other kernels were similar.

As sample size increases, performance initially improves and then flattens off. The point at which this occurs varies across kernels, but a sample size of 1000 is sufficiently conservative to reach the plateau for any of these kernels. Consequently, a sample size of 1000 is used for our experiments.

5.2. Methodology. The rest of the experiments in this section are conducted on the full set of 104 kernels. For each sampled configuration, *i*, in the upcoming results, *speedup* is calculated according to (1), where runtime_b is the runtime of the best performing configuration for that kernel from the random sample and runtime_i is the runtime of the configuration of interest. Therefore, the speedup of a given configuration from the random sample is always less than or equal to one, with higher values being better.

$$\text{speedup}_i = \frac{\text{runtime}_b}{\text{runtime}_i} \quad (1)$$

Sections 5.3 and 5.4 directly use the data obtained from performing a sample of 1000 configurations on each of the 104 kernels. Sections 5.5 and 5.6 perform additional experiments that require executing additional configurations.

5.3. High-Performing Configurations Are Few. Figure 3 shows a histogram of speedup values across all of the configurations; that is, each bar indicates how many optimization configurations fell in the given speedup bin. The majority of the configurations perform poorly, with only 1.3% and 1.0% of them achieving a speedup within 10% of the best performing configuration on the NVIDIA and AMD platforms, respectively. Conversely, a very large fraction of the configurations, 34.2% and 44.1% on NVIDIA and AMD, respectively, experienced more than a 10x slowdown relative to the best configuration. This shows that the optimization space is skewed towards poorly performing configurations. The AMD bars are smaller in general because fewer of the explored configurations were executable on the AMD platform due to a more restrictive maximum work-group size.

5.4. High-Performing Configurations Are Difficult to Find. We show that the best value for each optimization varies across different kernels. If a single value is always best for each optimization, then one can always set each optimization to its best value.

Figure 4 shows the number of times each possible value for each optimization was used in the best performing configuration. The graphs show that there is significant variation in the optimization values that give the best performance, particularly for WX, WY, WZ, and CZ where no individual value is best for more than 35 out of the 104 kernels on either platform. Data loading technique shows interesting differences between the two platforms, with local and image memory being favoured on the Titan, while all techniques other than image memory are roughly equally favoured on the R9 390. BX/VX shows less variation than the other optimizations, in large part due to the fact that it has a fixed value (of one) whenever vectorization is not used.

5.5. The Optimizations Interact. We show that the optimizations interact such that tuning the optimizations individually is not sufficient to find a good optimization configuration. For each kernel, we first determine the best value for each optimization while keeping the others held to the values used by the best configuration found by random sampling for that kernel. Then we combine the independently chosen optimizations to arrive at a final configuration.

A histogram of the speedup values of this approach across all 104 kernels is shown in Figure 5. The leftmost bar consists of configurations that are not even executable. On average, this approach achieves only 97% and 81% of the performance of the best sampled configuration on the Titan and the R9 390, respectively. That is, on average, it performs worse than its starting point. Thus, determining the best value for each optimization individually does not result in good performance; the optimizations must be examined together.

5.6. Good Configurations Are Not the Same across Kernels. Finally, we show that the best performing configuration for a kernel does not, on average, perform well across all kernels, thus demanding the auto-tuning of each kernel.

We run the best performing configuration of each kernel on the other kernels. Figure 6 shows a histogram of the resulting speedups. If the optimal configuration of each kernel worked best for all other kernels, all configurations would achieve a speedup of 1. The leftmost bar, once again, corresponds to configurations that are not even executable. Further, although many of the configurations perform well, and a few even exceed the performance of the best on a particular kernel, there is a significant tail to the histogram, indicating that performance can often be very poor. On average, these configurations achieve 69% and 63% of the performance of the best configuration on the Titan and the R9 390, respectively. This indicates that the best performing configuration cannot be learned for one kernel and used for other kernels.

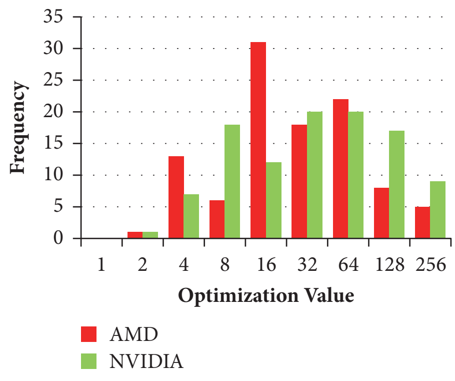
6. Auto-Tuning Strategy

6.1. Predicting Data Loading Technique. The use of local and image (i.e., texture or read-only) memory and vectorization, described in Section 4, all optimize how data is loaded from the GPU's memory hierarchy. We simplify the optimization space by only allowing data to be loaded in one of four ways: from global memory without vectorization, from global memory with vectorization, from local memory, or from image memory. As mentioned previously, we refer to this choice as the *data loading technique*. We determine this technique first because it has the largest impact on performance.

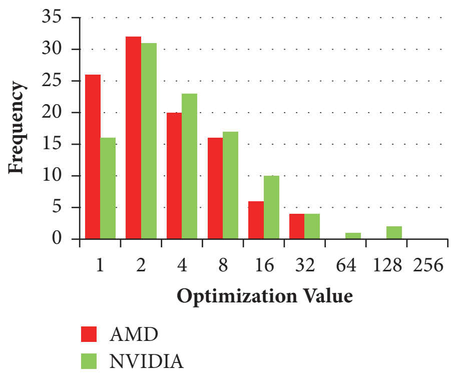
Vectorization has benefits as both a memory loading technique and a way of expressing additional computational parallelism. It is not possible to combine vectorization with the use of image memory. Nonetheless, it is possible to combine vectorization with loading data from local memory. However, we opt not to explore this combination. This is because, from the perspective of memory loading, both the use of local memory and vectorization allow us to load data from global memory in a coalesced manner. Given that using local memory alone achieves this same benefit, the only additional benefit of vectorization would be vectorizing our arithmetic operations. We forego this additional benefit and instead opt to reduce the size of the search space by limiting vectorization to loading data from global memory. (We present experimental data insupport of this decision in Section 8.3.2.)

The impact of the data loading technique depends to a large extent on the memory access patterns of the stencil computation. Thus, we hypothesize that the optimal data loading technique can be predicted using static program features, such as stencil size.

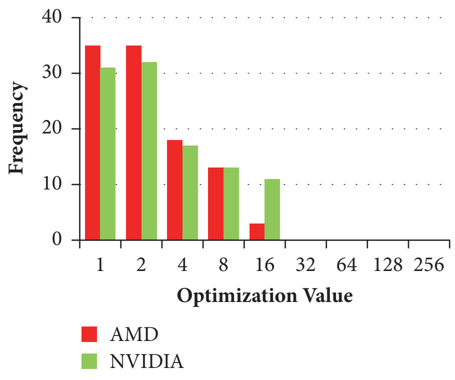
The mapping of static program features to optimal data loading technique is likely complex and platform-specific. We do not attempt to model it analytically. Instead, we use supervised machine learning [15] to predict the optimal data loading technique. We train a random forest [6] model to this end. The inputs to this model are stencil size, dimensionality, and density of a kernel as well as which dimension, if any, differs from the rest if the kernel is asymmetric (we refer to this dimension as the *unique dimension*). The output is the predicted optimal data loading technique for the kernel. We use Random Forests because of their ability to capture the discontinuities that often occur in the GPU performance space.



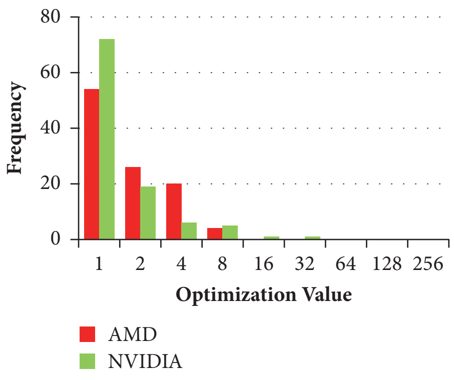
(a) WX



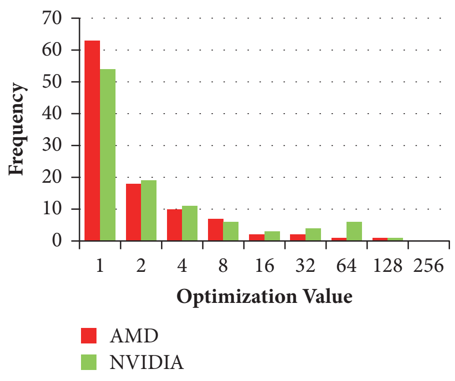
(b) WY



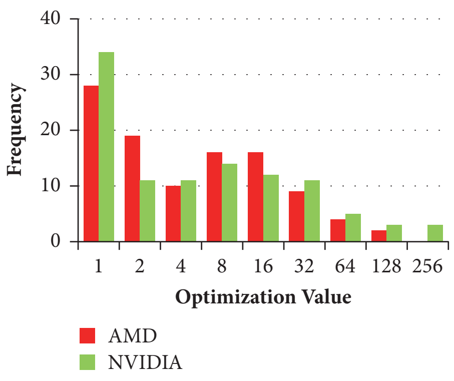
(c) WZ



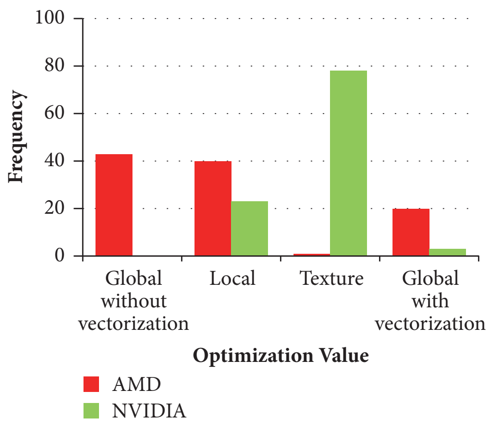
(d) CX



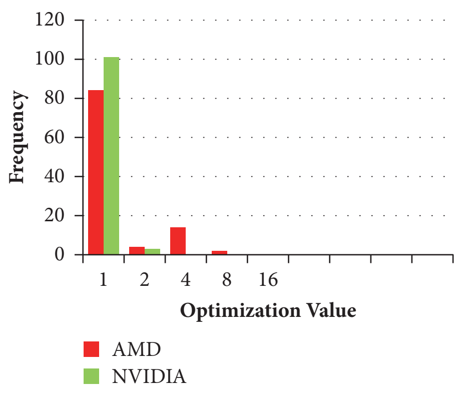
(e) CY



(f) CZ



(g) Data loading technique



(h) BX/VX

Figure 4: Distribution of best values for each optimization.

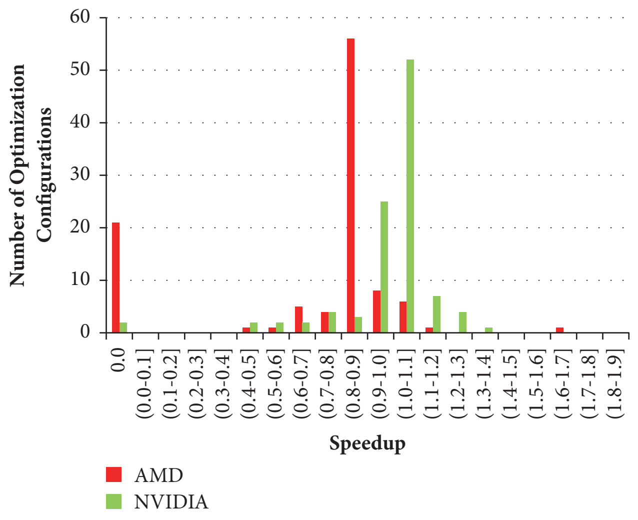


Figure 5: Speedup distribution when each optimization is explored independently.

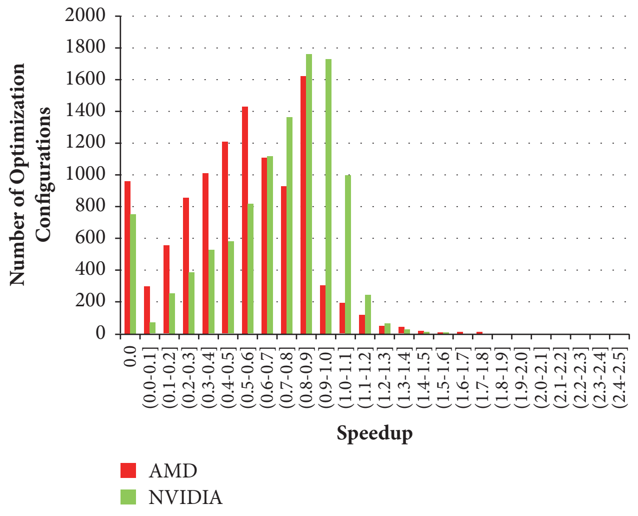


Figure 6: Speedup distribution of the best configuration for each kernel on all other kernels.

6.2. Grouping and Search Heuristics. Our auto-tuning heuristics further reduce the optimization space by breaking up the remaining optimizations into groups that are explored independently of each other. The values of the optimizations in each group are exhaustively explored, that is, the kernel is run for every possible value, while keeping the values of the other optimizations fixed. We refer to this process as *tuning* of the optimizations in question. Once the best values are found for these optimizations, they are fixed and the optimizations in the next group are tuned.

This approach has two key challenges. The first is how to group the optimizations, given that they are not independent of one another, as shown in Section 5.5. The groups are chosen so as to prioritize certain interactions amongst the optimizations over other interactions. Different groupings are explored in order to determine which interactions are most important. The second challenge is how to order the tuning of the groups. We tune the groups in order of their expected performance impact. We then mitigate the impact of group order by repeating the entire sequence of groups n times. The choice of n is important. If too small, then the heuristics will not have enough iterations to reach a local minima. If n is too large, the heuristics may evaluate extra configurations needlessly and take longer to run. Section 8.3.1 explores this parameter further.

We explore three grouping heuristics: two complementary ways of grouping the optimizations, by *dimension* and by *optimization*, as well as a hybrid approach between the two. In the first heuristic, all optimizations are explored for each dimension independently. In the second, each optimization is explored independently for all dimensions. Finally, the hybrid approach uses most of the groups of the group by dimension approach with the addition of a stage that tunes the work-group geometry.

Although we do not explore values of BY and BZ greater than one in our experiments (see Section 4.7), we show below how our heuristics would handle those parameters.

6.2.1. Group by Dimension. This heuristic considers all of the parameters that correspond to a particular dimension at once, regardless of the optimization they perform. Specifically, it

(1) sets WY = 1, WZ = 1, BY = 1, BZ = 1, CY = 1, and CZ = 1,

(2) tunes WX, BX, and CX,

(3) tunes WY, BY, and CY,

(4) tunes WZ, BZ, and CZ,

(5) repeats steps (2)–(4) n times.

We tune optimizations affecting the x dimension first, giving it highest priority. This is done because the innermost dimension is the one that impacts memory coalescing and most significantly benefits from caching. Since larger values are usually better for the work-group size in the dimension of coalescing, the initial values of WY and WZ are set to one so as to allow WX to be as large as possible. Similarly, BY, BZ, CY, and CZ are set to one initially to allow BX and CX to explore as large a range as possible before encountering significant register pressure.

6.2.2. Group by Optimization. This heuristic considers all of the parameters that correspond to a particular optimization at once, regardless of which dimension they affect. Specifically, it

(1) sets BX = 1, BY = 1, BZ = 1, CX = 1, CY = 1, and CZ = 1,

(2) explores WX, WY, and WZ while keeping BX, BY, and BZ fixed,

(3) tunes BX, BY, and BZ,

(4) tunes CX, CY, and CZ,

(5) repeats steps (2)–(4) n times; during each invocation of step (2), keep the following products constant: WX × CX, WY × CY, WZ × CZ.

By tuning the work-group size first over the thread merging factors, we prioritize overall parallelism level over register usage. Similarly, rather than holding CX, CY, and CZ constant in step (2), in all iterations after the first, $\mathrm{W_i} \times \mathrm{C_i}$ for i = X, Y, and Z is kept constant, thereby allowing the Cs to vary in a controlled manner. This product corresponds to the amount of work assigned to each work-group, while the Cs correspond to the amount assigned to each work-item. It is likely that the amount of work assigned to each work-group has a bigger impact on the total amount of available parallelism and thus this heuristic once again prioritizes this overall parallelism level over register usage.

Finally, we tune the block merging factor before the cyclic merge factor to ensure that there is as much register availability for block merging as possible because it is usually the more efficient way of merging threads as it allows for removal of duplicate reads through register sharing. (While in our current implementation we do not tune the block merging factors in the$y$ and z dimensions because of the OpenCL compiler limitation pointed out earlier inSection 4.7, we present our strategy assuming that they could be tuned.)

6.2.3. Hybrid Approach. This heuristic is the same as the group by dimension approach with the addition of a step that tunes the work-group geometry (step (5)). It

(1) sets WY = 1, WZ = 1, BY = 1, BZ = 1, CY = 1, and CZ = 1,

(2) tunes WX, BX, and CX,

(3) tunes WY, BY, and CY,

(4) tunes WZ, BZ, and CZ,

(5) tunes WX, WY, and WZ while keeping the following product constant: WX × WY × WZ,

(6) repeats steps (2)–(5) n times.

Step (5) here is similar to step (2) from the group by optimization heuristic; however it considers significantly fewer configurations. This is because it allows significantly less freedom in the chosen values of the optimization parameters being tuned. It consists of tuning the work-group geometry while keeping the total work-group size the same. This step is particularly important if the sizes of the work-group in each of the three dimensions interact strongly, as such an interaction can not be captured by the group by dimension heuristic.

6.3. Overall Strategy. Our overall strategy when auto-tuning a kernel is as follows: first use the machine learning model to predict the optimal data loading technique for the kernel and then apply our heuristic while fixing the data loading technique to the value predicted by the model. We evaluate three alternative heuristics and the optimal data loading technique can vary between them. Thus, we train a separate model for each heuristic.

7. Implementation

7.1. The Stencil Kernels. We desire a large, representative set of stencil kernels to both test our heuristics and train our machine learning models. Since there is no publicly available, large benchmark suite of such kernels, we generate a set of synthetic kernels to meet our needs. We use Genesis [16], a language and accompanying preprocessor for generating synthetic programs, to this end. Genesis allows us to generate a wide range of stencils that vary in a controlled manner using a single template file.

The elements of the kernel code that vary across kernels are implemented using Genesis constructs while the optimizations are implemented using C preprocessor directives. Thus, every generated program contains every optimization and before compiling a kernel, it is possible to select what optimizations to apply by setting various define statements. This allows optimization-related parameters to be known at compile time, allowing the compiler to perform as much simplification as possible.

We generate five stencil subtypes: dense, star, diamond, no-corners, and thumbtack, as depicted in Figure 1. For each subtype, we consider 1D, 2D, and 3D stencils variants. All the stencils act on 3D input and output arrays consisting of 256

Table 2: Breakdown of kernels by pattern, dimensionality, and radius.

Stencil pattern	Dimensionality			Radius					
	1D	2D	3D	0	1	2	3	4	5
Star	0	15	5	0	4	4	4	4	4
Thumbtack	0	0	15	0	3	3	3	3	3
Diamond	0	12	4	0	0	4	4	4	4
No-corners	0	12	5	0	1	4	4	4	4
Dense	16[†]	15	5	1	7	7	7	7	7

[†] We classify these 16 kernels as dense, but since all patterns are the same in one dimension, this choice is arbitrary.

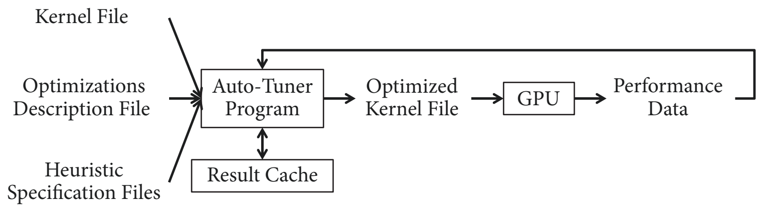


Figure 7: The auto-tuner flow.

elements in each dimension. For the 1D and 2D stencils, we also use 3D input and we consider each possible orientation of the stencil. This is because each orientation has different memory access characteristics. The stencil radii range from 0 to 5. We generate 104 different kernels in this way, as summarized in Table 2.

The kernels perform a weighted sum of their input elements using nonzero weights that are randomly chosen during program generation. The weights are thus known at kernel compile time. We assume that boundary data is copied into place beforehand. All the kernels are run with single-precision floating point input data that is randomly generated.

Our synthetic stencil kernels subsume many real world stencil applications. For example, the Gaussian blur filter [17], a common image processing application, is a 2D dense stencil with the same computational properties as those modeled in this work. Our synthetic stencils enrich our evaluation by ensuring that we have stencils with a diversity of properties. One exception to this diversity is the computational intensity of the stencils. We address this limitation in Section 8.3.8.

7.2. The Auto-Tuner. Figure 7 shows our auto-tuner's flow. It takes as input a kernel, an optimizations description file, and a set of heuristic specification files. The optimizations description file details the optimizations that are present in the kernel, their ranges of allowable values, and a few other important characteristics of each optimization. Each heuristic specification file describes what optimizations to vary and which to keep constant in each step of a heuristic.

The auto-tuner copies the input data from the host to the GPU before the first configuration is tested. If a desired configuration has already been compiled, its runtime is retrieved from a cache of past results; otherwise the auto-tuner sets various optimization-related parameters in the code and compiles the kernel using the OpenCL API. If the kernel can be launched on the device, the OpenCL binary is run on the GPU four times and the average runtime of the last three runs is recorded.

Our approach requires a recompile for every unique optimization configuration, even if configurations differ only in runtime parameters such as the work-group size. This allows us to make these values compile time constants rather than getting them at runtime using OpenCL API functions. This maximizes the compiler's ability to optimize by constant folding, constant propagation, and dead code elimination.

For machine learning training, we run each heuristic with each data loading technique. We then train and test a model for each heuristic using leave-one-out cross-validation [15]. This method of validation ensures that a model is only tested with kernels that were not used to construct it.

7.3. Heuristic Specification Files. The auto-tuner supports a generic heuristic specification file syntax that can be used to describe a wide range of heuristics. This syntax allows optimizations to be assigned ranges of values and to be assigned sampled values from ranges. It also allows for the use of mathematical expressions to calculate optimization values or ranges from the values of other optimizations, the values of the best configuration currently discovered, and the input data size. All of the experiments done in this work were conducted using heuristic specification files.

Algorithm 3 shows the heuristic specification file for the group-by-dimensions heuristic in the case where local memory was chosen as the data loading technique. In this case, $n = 1$ (i.e., one additional iteration over the groups is performed). Each line of this file corresponds to one of the steps from Section 6.2.1.

7.4. Tuning a New Stencil. The use of our auto-tuner framework for a new stencil entails a number of steps:

(1) Determining the machine learning input features (described in Section 6.1) for the new stencil. These

```
BX=1,BY=1,BZ=1,WY=1,WZ=1,CY=1,CZ=1,USE_LOCAL_MEMORY_INPUT=1,VX=1,
      STORE_ARRAYS_AS_IMAGES=0,WX=1:NX:∗2,CX=1:NX/WX:∗2;
WY=1:NY:∗2, CY=1:NY/WY:∗2;
WZ=1:NY:∗2, CZ=1:NZ/WZ:∗2;
WX=1:NX:∗2, CX=1:NX/WX:∗2;
WY=1:NY:∗2, CY=1:NY/WY:∗2;
WZ=1:NZ:∗2, CZ=1:NZ/WZ:∗2;
```

ALGORITHM 3: The heuristic specification file for the group-by-dimensions heuristic using local memory.

features can be determined readily through inspection of the code.

(2) Running the machine learning model for the heuristic of choice to determine the best data loading technique.

(3) Parameterizing the kernel for the optimizations described in Section 4. At present, and without compiler support, this is where most of the work of using our approach lies.

(4) Running the auto-tuner with the heuristic specification file for the desired heuristic and the desired data loading technique.

The majority of the effort required to apply our approach to new stencils comes from implementing our optimizations in a parameterized way. This effort could be eliminated entirely through compiler support.

8. Evaluation

This section gives our evaluation of the auto-tuning strategy discussed in Section 6.

8.1. Reference Approaches. We compare our heuristics to 3 reference approaches: random sampling, expert search, and an oracle. We describe these approaches below.

(i) *Random sampling* (`Rand`): this approach corresponds to running the first stage of the interestingness experiment described in Section 5, i.e., running 1000 configurations, and selecting the best performing one.

(ii) *Expert search* (`ExpS`): this approach performs exhaustive, empirical auto-tuning but only on a restricted subset of the space chosen by an expert. We restrict the space as follows. VX must be less than or equal to 4 since this length often results in good performance. WX must be greater than or equal to 32 since this maximizes coalescing when it is present. WY × CY must be less than or equal to 4 and WZ × CZ must be less than or equal to 4. Both these values allow for a sufficient number of work-groups and avoid loss of parallelism. This approach represents the one used by most existing stencil auto-tuning work in the literature (described in Section 9). In our experience, it is also similar to how programmers hand-tune kernels. Thus, we use it as a reference that represents both existing approaches and hand-tuning.

(iii) *Oracle*: there is an oracle-based approach for each of our heuristics. Each consists of running the heuristic with every data loading technique and then taking the best configuration found across all of them. These bounds are “oracles” for the machine learning models. That is, they effectively give upper bounds on the speedups that could be achieved with perfect machine learning models.

8.2. Evaluation Platforms and Metrics. Our results are gathered on an NVIDIA GTX Titan GPU and an AMD Radeon R9 390. The host is an Intel i7 960 with 6 GB of RAM running Ubuntu 14.04, NVIDIA driver version 352.41, AMD driver version 15.201, and the OpenCL 2.0 ICD loader library from the Khronos group. Both GPUs we use have generic read-only caches rather than multidimensional-locality-exploiting texture caches, which we use for image storage. We auto-tune a single time iteration of the stencil (i.e., a single invocation of the kernel) and thus do not perform the swap of the input and output arrays described in Section 4.6. Our use of OpenCL allows the use of the same kernels across both GPUs with no change. The machine learning models are created using Weka 3.6 [18] with its default random forest parameters.

Each of the machine learning models is assessed using two metrics. The first is its *absolute accuracy*, i.e., on what fraction of the kernels it predicts the correct data loading technique. The second is its *penalty-weighted accuracy*, defined as the average speedup of the heuristic using the model relative to its corresponding oracle, averaged over all kernels. Thus, the penalty-weighted accuracy reflects the penalty of a mispredicted loading technique on performance.

The performance of the heuristics is assessed with two metrics. The first is the runtime of the best configuration discovered by a heuristic. This time is normalized by calculating the speedup with respect to the best configuration found by `Rand`. The geometric mean of speedups is used to report average speedup across the kernels. We use the geometric mean instead of the arithmetic mean because it is more meaningful when aggregating ratios [19]. The second metric is the time the heuristic takes to get to that configuration, measured by the number of configurations compiled, the total time spent compiling these configurations, and the runtime of these configurations on the GPU. The runtime of each configuration is that of a single run of a kernel, i.e., a single iteration of the time loop, averaged over three consecutive runs.

Table 3: Performance of heuristics and reference approaches.

	Speedup		Configs		Compile time (s)		Runtime (s)		Best count	
	NV	AMD	NV	AMD	NV	AMD	NV	AMD	NV	AMD
Baselines										
ExpS	1.07	1.21	1311	1066	937.48	246.13	27.02	5.18	24	22
Rand	1.00	1.00	321	210	344.71	52.75	37.93	9.03	2	2
Oracles										
Dimensions	1.09	1.17	720	711	712.84	167.39	96.15	24.39	22	11
Optimizations	1.15	1.32	1768	1526	1076.60	339.25	103.39	25.80	62	42
Hybrid	1.14	1.34	992	1018	1062.82	242.38	104.85	26.73	58	55
Heuristics										
Dimensions	1.07	1.13	181	191	55.24	36.01	20.85	4.23	19	11
Optimizations	1.14	1.28	502	357	136.28	77.45	20.64	5.26	57	36
Hybrid	1.12	1.32	266	266	88.35	53.67	21.51	5.45	54	48

8.3. Experiments

8.3.1. Determining a Good Value for n. The parameter n determines how many times the heuristics iterate over their optimization groups. Increasing n allows a heuristic to potentially find a better performing optimization configuration at the cost of more compiles and more runtime. This tradeoff is only possible up until the point that the heuristic reaches a local minima and no longer examines any further unique configurations.

In order to determine a good value for n, the group-by-dimensions heuristic was run on a subset of the kernels for various values of n. The seven kernels of dimensionality three and radius two were chosen for this experiment as they represent a reasonable cross-section of the overall kernel set while running in a reasonable amount of time.

We found that there was a large benefit from adding a second iteration to the heuristic, i.e., going from $n = 0$ to $n = 1$. There was a very slight improvement from $n = 1$ to $n = 2$. After that, there was no benefit to additional iterations. In terms of the results on the individual kernels, it took only one iteration to reach a local minima on four of the seven, two iterations for two of them, and three iterations for only one. As one of the test kernels did require it to achieve a local minima, in order to be conservative in the rest of the results presented, a value of $n = 2$ will be used.

8.3.2. Auto-Tuning Strategy Performance. The speedups, number of unique configurations compiled, compile times, and GPU runtimes (in seconds) for all heuristics and reference approaches on both NVIDIA (NV) and AMD platforms, averaged over all kernels, are shown in Table 3. The table also shows "best count", the number of times each strategy found the best performing configuration amongst those explored. Note that the best counts for the oracles are necessarily greater than or equal to those of our strategies because they look at a superset of the configurations of our strategies. Further, the best count column total for the baselines and the oracles is more than 104 because for some kernels more than one strategy finds the best configuration (i.e., they find the same configuration).

A number of observations can be made from this table. First `ExpS` outperforms `Rand`, particularly on the AMD platform, both in speedup and in best count. It explores significantly more configurations in order to do so but requires less device-side runtime as these configurations are generally better performing. This impact is more pronounced on the AMD platform because fewer configurations are explored on this platform due to the fact that a larger fraction of the 1000 configurations chosen at random were unexecutable on this platform. This is because the Radeon R9 390 has a much smaller maximum work-group size than the GTX Titan. As such, the baseline performance on the AMD card is lower and thus all strategies have better relative performance.

Secondly, the machine-learning-based heuristics, with the exception of the group-by-dimensions approach on the AMD platform, are able to achieve equal or higher speedups than both of those reference approaches on both platforms. These heuristics do this while considering significantly fewer configurations than expert search and a comparable number of configurations to random sampling. Furthermore, the amount of compile time they take per configuration is lower because they waste less time than `Rand` considering invalid configurations which are not determined to be invalid until after compilation. They also require about the same amount of device-side runtime as the expert search strategy. In particular, the hybrid approach performs very well, requiring 71% and 4% less total tuning time than `Rand` on NVIDIA and AMD, respectively, while finding best configurations that are 12% and 32% better on average. It takes 89% and 76% less time than `ExpS` (on NVIDIA and AMD, respectively) while finding best configurations that are 5% and 9% better on average. The poor performance of the group-by-dimensions heuristic on the AMD platform is largely a result of the reduced maximum work-group size on this platform. This restriction combined with the nature of this heuristic prevents it from exploring a range of work-group shapes with large values in the y and z dimensions. Since the hybrid approach incorporates a stage that tunes the work-group shape independently of size, it does not have this problem even though the heuristics are the same otherwise.

Table 4: Breakdown by data loading technique (NVIDIA).

Data loading technique	Speedup			Best count		
	Dims	Opts	Hybrid	Dims	Opts	Hybrid
Global without vectorization	0.54	0.54	0.54	0	0	0
Global with vectorization	0.45	0.45	0.45	3	1	2
Image	1.00	1.05	1.04	13	40	36
Local	0.76	0.87	0.86	6	21	20

Table 5: Breakdown by data loading technique (AMD).

Data loading technique	Speedup			Best count		
	Dims	Opts	Hybrid	Dims	Opts	Hybrid
Global without vectorization	0.95	1.02	1.01	4	12	11
Global with vectorization	0.87	0.85	0.92	6	4	17
Image	0.56	0.58	0.58	0	0	0
Local	0.93	1.18	1.20	1	26	27

Table 6: Machine learning model accuracies.

	Absolute accuracy		Penalty-weighted accuracy	
	NVIDIA	AMD	NVIDIA	AMD
Dimensions	0.87	0.81	0.98	0.96
Optimizations	0.91	0.80	0.99	0.97
Hybrid	0.88	0.88	0.99	0.98

Comparing the dimensions and optimizations heuristics, we see that the optimizations approach is able to achieve a substantially higher speedup especially on the AMD platform but at the cost of two to three times the number of configurations and significantly more runtime. The hybrid approach seems to get the best of both worlds. It achieves the high performance of the group-by-optimizations approach while only costing a slight premium over the group-by-dimensions heuristic. Thus, we believe that the hybrid heuristic provides the best tradeoff between the various metrics. With that said, the group-by-dimensions heuristic could be useful for users that require a very short tuning time.

Tables 4 and 5 give insight into why our strategies perform well. They summarize the speedups and best counts achieved by our heuristics for each data loading technique (i.e., without the use of the machine learning models). None of the individual loading techniques is best all of the time, and with the exception of local memory on the AMD platform, none of them has a speedup much larger than `Rand`. However, `Rand` rarely finds the best configuration of all of the strategies as evidenced by its best count of 2 in Table 3 on both the NVIDIA and AMD platforms. Conversely, the individual data loading techniques, although worse on average, have much higher best count values; when they do well, they do very well. By combining the strengths of all of these data loading techniques, the machine-learning-based strategies are able to achieve much higher best counts. They reach greater average speedups than `Rand` and `ExpS` with fewer configurations by choosing the right data loading technique for each kernel.

Tables 4 and 5 show that image data loading technique does dominate over the other techniques for the NVIDIA platform but not for AMD. However, this does not mean that there is no value to the other data loading techniques on NVIDIA. Our hybrid oracle shows a speedup of 1.14 and our hybrid predicted achieved a speedup of 1.12 relative to the hybrid image-only of 1.04. This indicates that there is a benefit to not choosing image memory for every kernel. The fact that one outcome is more likely than the others does not necessarily mean that the model can be made simpler, merely that the model should be predicting that outcome more often.

The accuracies of the machine learning models are shown in Table 6. The absolute, count-based accuracies are reasonable but not excellent; however, the penalty-weighted accuracies are very high. This indicates that although the models do not always predict correctly, they predict correctly for those kernels where it makes a big difference. In other words, when they are incorrect, their mistakes have little impact on performance.

We also visualize the distribution of speedup across the 104 stencil kernels; Figure 8 shows a histogram of speedup for the hybrid strategy on both the NVIDIA and AMD platforms. On both platforms, the strategy rarely underperforms `Rand`, with only 12 and 6 kernels having a speedup less than one on the NVIDIA and AMD platforms, respectively. The worst performing kernel on NVIDIA has a slowdown of 12% while

Table 7: Results for the restricted heuristics.

	Speedup		Configs		Compile time (s)		Runtime (s)	
	NVIDIA	AMD	NVIDIA	AMD	NVIDIA	AMD	NVIDIA	AMD
Hybrid	1.12	1.32	266	266	88.35	53.67	21.51	5.45
Hybrid restricted	1.13	1.30	154	108	35.90	22.98	2.71	0.38

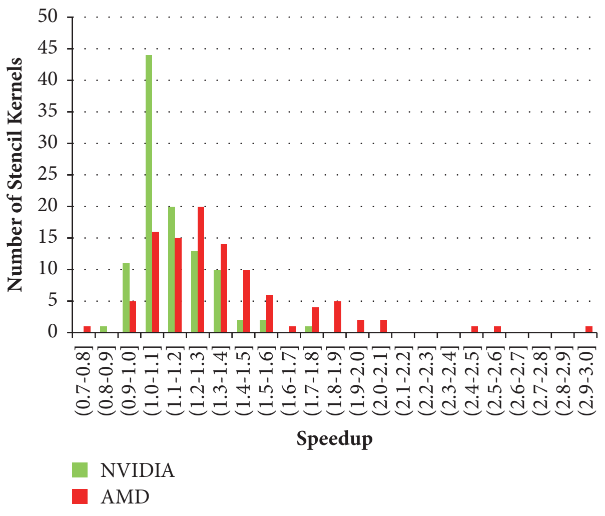


Figure 8: Speedup distribution of the hybrid strategy.

the worst on AMD has a slowdown of 24%. Both platforms have a large number of kernels with a moderate speedup and both platforms have a long tail towards positive performance. The best speedup on NVIDIA was 72%, while the best on AMD was 196%. Overall, these are desirable characteristics for an auto-tuning strategy; the most common outcome is an improvement over the baseline with large increases in performance possible and decreases in performance uncommon and, when present, small.

Finally, we examine the impact of our decision not to explore vectorization concurrently with the rest of the data loading techniques (Section 6). We examine kernels for which local memory was the best data loading technique. The hybrid heuristic using local memory achieved speedups of 1.25 and 1.44 (on NVIDIA and AMD, respectively). This is in contrast to only 0.34 and 0.54 for the same kernels with the same heuristic using vectorization. Conversely, on kernels for which vectorization was the best data loading technique, vectorization achieves speedups of 1.07 and 1.49 (on NVIDIA and AMD). For these kernels, local memory achieves only 0.67 and 1.05. These results suggest that the types of kernels on which one technique performs well are different from the kernels on which the other techniques perform well. This is despite the fact that local memory was the better technique on average across all kernels for both platforms. Thus, this suggests that it was reasonable not to explore vectorization concurrently with other data loading techniques.

8.3.3. Computational Throughput. On the NVIDIA GTX Titan, our kernels tuned by the hybrid heuristic achieve throughputs ranging from 30 to 646 GFLOPS with an average of 411 GFLOPS. On the AMD R9 390, the range was from 39 to 2180 GFLOPS with an average of 855 GFLOPS. The lowest throughput on both platforms occurs on the radius 0 kernel (i.e., multiply every element in an array by a constant). The highest occurs on a dense 3D stencil of large radius. The lower throughputs on the sparser kernels reflect that these kernels are more memory-bound than compute-bound.

8.3.4. Restricting the Optimization Space. We can use expert knowledge of the optimization space to further limit the exploration performed by the heuristics and thus reduce the number of configurations they consider. We force the work-group size in the x dimension to be at least 16 in order to exploit memory coalescing. We also prohibit the work-group sizes in the y and z dimensions from exceeding 32, as sizes beyond that are rarely beneficial. We can afford to impose looser restrictions than `ExpS` because our strategy has already significantly reduced the optimization space size. The results of applying these restrictions to our best performing heuristic, the hybrid heuristic, are shown in Table 7. There is a significant reduction in the number of configurations considered and an even more significant reduction in runtime. This is because the excluded configurations are generally the worst performing ones. Speedup remains relatively unchanged because the new heuristic is not able to explore any configurations that the old one could not.

This shows that our strategy complements expert search approaches as the two strategies can effectively be used together.

8.3.5. Variable Work-Group Size. In order to allow the compiler to optimize as much as possible, the work-group size is fixed at compile time. This dramatically increases the number of compilations made. We justify this approach by considering variants of the kernels in which the work-group size is not fixed. The hybrid heuristic is then run on these modified kernels. Once the heuristic has reached its final configuration, that configuration is rerun using the variant of the kernel with a fixed work-group size.

The results of this experiment on both platforms are shown in the second row of Table 8. The first row of this table is the performance on the original kernels, copied from Table 3. Relative to the same heuristic applied to the fixed work-group size kernels, this approach results in a dramatically lower average speedup of only 0.94 and 1.04 on NVIDIA and AMD, respectively. It does, however, require dramatically less compile time as it does not require a recompile for each configuration. There is an increase in device-side runtime due to the fact that many of the explored configurations perform worse.

8.3.6. Array Size. This experiment seeks to determine the effect of array size on the interestingness of the space and the performance of the strategies. The hybrid strategy was run on arrays ranging in size from 32^3 to 512^3 using the machine learning models trained previously on the original, 256^3 kernels. To save time, this experiment was only run only on the seven kernels used in Section 8.3.1.

Table 9 shows the performance of the strategy, the relevant oracle, and random sampling on these different sizes. As long as data size remains above some minimum threshold, the machine learning model continues to perform well and the performance of the strategy matches that of the oracle. As size decreases however, the oracle performance itself decreases and thus the performance of the strategy does as well. Eventually the machine learning model no longer predicts correctly. Both of these issues are likely because the space itself is less interesting for these small array sizes. For example, global memory becomes more favoured as the input size decreases because the L1 and L2 caches of the GPU are able to better address the memory needs of the kernel.

8.3.7. Array Shape. The hybrid strategy was run on kernels with input sizes of 1024×1024 in x and y and ranging in size from 8 to 128 in the z dimension using the machine learning models previously trained on the 256^3 kernels. The total number of array elements is always more than 128^3, the point below which performance degraded in the previous experiment. This removes the impact of input size.

The results are shown in Table 10 in the same format as Table 9. What is most noticeable is the consistency in the machine learning models' performance. On NVIDIA, for all but the smallest value of z, the machine learning model predicts correctly, allowing the strategy to achieve close to the oracle performance. On AMD the model's accuracy is also consistent, although not as good, but the performance of our strategy decreases as the array becomes less symmetric. This shows that our machine learning models are fairly insensitive to array shape, but for some platforms it may be necessary to incorporate array shape into the machine learning model.

8.3.8. Varying Computational Intensity. One limitation of our synthetic kernels is that they all have a single add and a single multiply per loaded input element. In this regard, they underrepresent some real applications that have higher computational intensities. In this section, we show that our strategies remain effective with higher computational intensities.

We regenerate our synthetic stencils such that the number of additions and multiplications per read element is set to either 1 or 2. To prevent the compilers from optimizing away these multiplications or additions through constant folding, we multiply/add each read input data element with itself as necessary to increase the multiplication/addition count. This results in a total of 416 kernels, which we use to train a new machine learning model with the number of multiplications and the number of additions as new input features. The hybrid strategy is rerun for these kernels and the results are reported only for the AMD GPU for brevity.

Table 11 shows the performance of the 4 strategies: expert search, random sampling, hybrid oracle, and hybrid heuristic. Similar conclusions to those in Section 8.3.2 can be drawn here regarding the relative performance of the strategies. The accuracy of the machine learning model remains high at 0.89 absolute and 0.98 penalty-weighted. Furthermore, the throughput increases as we would expect, reaching an average of 1072 GFLOPS with a peak of 2619 GFLOPS (on a 2D dense stencil of radius 5). The performance of the heuristic does not vary significantly with the number of multiplications or additions. There is a slight increase in performance as arithmetic intensity increases, but the expert search strategy exhibits the same increase. This may indicate that the optimization space becomes more interesting on the stencils with higher arithmetic intensity, leading to a lower relative performance for random sampling. This validates the suitability of our strategies when the intensity of stencil computations varies.

9. Related Work

There is a large body of work dealing with automatic performance tuning of stencil codes, on both multicores and GPUs. Some of this work exhaustively searches the optimization space [20] while some of it uses expert knowledge to first limit the optimization space and then perform exhaustive search on that subset of the space [21–25]. In contrast, our work does not require expert knowledge of the platform or the application. It does not limit the values that are searched for each optimization, rather which optimizations are searched concurrently. Furthermore, we show that our strategy complements approaches that utilize expert knowledge. Specifically, we show in Section 8.3.4 that incorporating expert knowledge into our strategy improves performance even further.

Table 8: Results for variable work-group size kernels.

	Speedup		Configs		Compile time (s)		Runtime (s)	
	NVIDIA	AMD	NVIDIA	AMD	NVIDIA	AMD	NVIDIA	AMD
Fixed size	1.12	1.32	266	266	88.35	53.67	21.51	5.45
Variable size	0.94	1.04	49	68	14.92	14.32	42.15	6.09

Table 9: Heuristic performance on various input sizes.

	Data size per dimension									
	NVIDIA					AMD				
	32	64	128	256	512	32	64	128	256	512
Speedup										
Hybrid oracle	1.03	1.04	1.15	1.18	1.19	0.98	0.93	1.12	1.07	1.36
Hybrid	0.97	1.04	1.15	1.18	1.19	0.85	0.91	1.11	1.06	1.39
Random sample	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Correct predictions	2	7	7	7	7	3	3	6	6	6

Table 10: Heuristic performance on various input shapes.

	Z size									
	NVIDIA					AMD				
	8	16	32	64	128	8	16	32	64	128
Speedup										
Hybrid oracle	1.11	1.20	1.20	1.17	1.16	1.27	1.26	1.34	1.52	1.50
Hybrid	1.08	1.20	1.20	1.17	1.16	0.95	1.10	1.26	1.48	1.50
Random sample	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Correct predictions	5	7	7	7	7	4	4	4	4	6

Some existing GPGPU auto-tuning works have used intelligent, nonexhaustive strategies [12–14, 20, 26]. However, all of this work (with the exception of Tang's which develops an analytical model to approximate performance [26]) focuses exclusively on a single optimization. In contrast, we consider multiple optimizations concurrently and our strategy takes into account the interactions amongst these optimizations.

One common theme with all of the works mentioned so far is the focus on a handful of applications. In contrast, we use a large number of programs that were synthetically generated to have a wide range of stencil patterns and sizes. These synthetic stencils subsume many real world stencils and thus enrich our evaluation by ensuring that we have stencils with a diversity of properties. Some auto-tuning works have made up for this deficiency though by considering different inputs [27, 28], which essentially expands the program space in an alternative manner.

There are several approaches that use statistical and machine learning models to partition and prune the optimization configuration space, e.g., recursive partitioning regression trees [29] and kernel canonical correlation analysis [5]. These approaches require running the program of interest in order to train their models while our machine learning models can be trained ahead of time because they are trained on other programs.

There has also been work on auto-tuning frameworks that employ a host of generic (i.e., non-stencil-specific and non-GPU-specific) search techniques to manage large search spaces. One example is MIT's OpenTuner [30], which uses strategies such as Nelder-Mead, differential evolution, and gradient descent to search through the optimization space. Another example is the PATUS framework [31], which incorporates one domain specific language to specify stencils and a second to specify the parallelization approach. It uses exhaustive search or generic approaches to auto-tuning, such as multi-run Powell search or evolutionary search. In contrast to these frameworks, our contribution is not a framework for search, but rather a strategy that exploits both stencil and GPU specific knowledge to make the search efficient.

A comparison of the performance of our strategy to those used by the previous auto-tuning frameworks is certainly desirable. For example, it is impractical to make a direct comparison with OpenTuner because it has little support for GPU/OpenCL. In particular, it lacks the ability to express interoptimization constraints that often exist on GPUs (e.g., the restriction on the product of W, B, and C mentioned in Section 4.7). Not observing such constraints can result in grossly increased auto-tuning times because of the many infeasible configurations that must be examined and compiled. While some attempts have been made to address such constraints in OpenTuner [32, 33], it is necessary to modify OpenTuner to express the constraints that exist in our work. This makes it difficult to make a fair comparison to OpenTuner.

Table 11: Performance for varying computational intensity.

Strategy	Speedup	Configs	Compile time (s)	Runtime (s)
ExpS	1.23	1066	250.30	5.36
Rand	1.00	210	54.19	9.57
Hybrid oracle	1.32	1018	245.35	27.70
Hybrid heuristic	1.32	253	52.71	5.23

Nonetheless, we also make a meaningful comparison to existing work throughout the results section, albeit a nondirect one. We compare the performance of our strategy to an expert search that uses domain specific knowledge to sufficiently narrow the space for exhaustive search, often resulting in high-performing configurations that would be similar to those obtained by aforementioned works. Our strategy results in as good or better configurations. Further, our strategy is able to achieve computational throughput that is similar to or better than that achieved by other strategies [31, 34], as was described in Section 8.3.3.

The work presented here extends the authors' earlier work [35] in that it describes a new hybrid approach heuristic, expands the machine learning model input features, validates the work on a second GPU platform, and provides new experiments to demonstrate the computational throughput of the approach as well as the impact of variable workgroup size, variable array sizes, variable array shapes, and variable computational intensity.

Finally, there has been extensive work on auto-tuning of kernels other than stencils [36–39]. These works are too numerous to list exhaustively. In general, the size of this field demonstrates the importance of auto-tuning in extracting performance from GPU platforms.

10. Conclusions

We presented a strategy for automatic performance tuning of stencil computations on GPUs when applying common optimizations. Our strategy uses machine learning to determine the best approach to memory loading and then explores the remaining optimizations in groups. We presented three possible ways of grouping the optimizations so as to reduce intergroup dependencies. We synthetically generated 104 OpenCL kernels and evaluated our strategy both in terms of the time needed for auto-tuning and in terms of the quality of the best configuration obtained. We showed that our best heuristic achieves a reduction in total exploration time relative to expert search (by 89% and 76% on an NVIDIA GeForce GTX Titan and an AMD Radeon R9 390, respectively) while finding better performing configurations (by 5% on the Titan and 9% on the R9 390). Compared to random sampling, we can achieve much higher performance (12% and 32% on NVIDIA and AMD, respectively) in less or comparable time (71% and 4% less on the Titan and the R9 390, respectively).

Our study has shown a successful strategy for the auto-tuning of stencils on GPUs that combines a machine learning model with heuristic search. The effectiveness of our strategy was demonstrated by comparing it to an "expert search" strategy that is representative of other approaches for auto-tuning stencils as well as to random sampling. While our study has considered only synthetic kernels, we believe that these kernels subsume many real applications and are thus representative of several classes of stencils used in practice. Further, we demonstrated the effectiveness of our strategies across stencil shapes, input size and shape, and varying computational intensity. Nonetheless, our study is limited to stencils with single input/single output arrays and without loop-carried dependence. Further it also does not consider optimizations to the time loop of a stencil. Extending the study to these other types of stencils and to include optimizations to the time loop are potential directions for future work. Other directions include exploring other machine learning models and other target GPU platforms.

Appendix

A. Best Configurations Found on NVIDIA Platform

This appendix gives the best configurations found by the hybrid strategy on the NVIDIA platform. Dims stands for the dimensionality of the stencil; Unique Dim refers to the unique dimension of the stencil, as defined in Section 6.1, "vec" stands for global memory with vectorization, and "global" stands for global memory without vectorization. The results in Table 12 further validate our interestingness study and show that no single configuration is always best across the stencils.

B. Best Configurations Found on AMD Platform

This appendix gives the best configurations found by the hybrid strategy on the AMD platform. Again, Dims stands for the dimensionality of the stencil; Unique Dim refers to the unique dimension of the stencil, as defined in Section 6.1, "vec" stands for global memory with vectorization, and "global" stands for global memory without vectorization. The results also show that no single configuration is always best across the stencils (see Table 13).

Table 12

Radius	Type	Dims	Unique Dim	BX/VX	WX	WY	WZ	CX	CY	CZ	Data loading technique
1	thumbtack	3	x	1	128	4	1	1	32	1	image
1	thumbtack	3	y	1	128	2	1	1	1	16	image
1	thumbtack	3	z	1	128	2	1	1	8	1	image
2	thumbtack	3	x	1	128	2	4	1	1	64	image
2	thumbtack	3	y	1	64	1	4	1	1	1	image
2	thumbtack	3	z	1	128	2	2	1	1	64	image
3	thumbtack	3	x	1	8	2	8	1	1	1	image
3	thumbtack	3	y	1	64	1	2	1	1	1	image
3	thumbtack	3	z	1	64	1	2	1	1	1	image
4	thumbtack	3	x	1	64	4	2	1	1	32	image
4	thumbtack	3	y	1	64	2	2	1	1	4	image
4	thumbtack	3	z	1	128	1	1	1	1	1	image
5	thumbtack	3	x	1	128	4	1	1	1	16	image
5	thumbtack	3	y	1	128	1	2	1	1	4	image
5	thumbtack	3	z	1	128	2	1	1	16	1	image
0	dense	1	none	4	8	64	2	1	1	1	vec
1	dense	1	x	4	64	4	1	1	1	1	vec
1	dense	1	y	1	128	1	2	1	4	4	image
1	dense	1	z	1	128	4	1	1	8	1	image
1	dense	2	x	1	128	4	1	1	16	1	image
1	dense	2	y	1	128	2	1	1	1	32	image
1	dense	2	z	1	128	2	1	1	16	1	image
1	dense	3	none	1	32	2	2	2	1	16	local
2	dense	1	x	1	128	1	2	1	8	2	image
2	dense	1	y	1	128	2	1	1	32	1	image
2	dense	1	z	1	128	2	2	2	64	1	image
2	dense	2	x	1	16	4	2	1	2	32	local
2	dense	2	y	1	64	1	2	2	1	32	local
2	dense	2	z	1	32	4	1	2	2	1	local
2	dense	3	none	1	32	8	1	1	1	64	local
3	dense	1	x	1	32	1	8	1	16	1	image
3	dense	1	y	1	64	8	1	1	32	1	image
3	dense	1	z	1	16	2	32	1	8	4	image
3	dense	2	x	1	16	8	2	1	1	64	local
3	dense	2	y	1	64	1	2	2	1	16	local
3	dense	2	z	1	32	4	1	1	4	1	local
3	dense	3	none	1	32	8	2	1	1	32	local
4	dense	1	x	1	64	1	2	1	1	1	image
4	dense	1	y	1	64	4	1	1	32	1	image
4	dense	1	z	1	8	1	16	1	1	8	image
4	dense	2	x	1	8	16	2	1	1	64	local
4	dense	2	y	1	64	1	2	1	1	32	local
4	dense	2	z	1	32	8	1	2	2	1	local
4	dense	3	none	1	64	8	2	1	1	64	local
5	dense	1	x	1	64	1	2	1	1	1	image
5	dense	1	y	1	32	8	1	1	32	1	image
5	dense	1	z	1	16	4	8	1	1	16	image
5	dense	2	x	1	8	8	4	1	2	32	local
5	dense	2	y	1	64	1	2	1	1	16	local
5	dense	2	z	1	32	8	1	2	2	1	local
5	dense	3	none	1	4	4	32	1	1	1	image
1	star	2	x	1	128	4	1	1	8	1	image
1	star	2	y	1	128	4	1	1	8	1	image

Table 12: Continued.

Radius	Type	Dims	Unique Dim	BX/VX	WX	WY	WZ	CX	CY	CZ	Data loading technique
1	star	2	z	1	128	2	1	1	16	1	image
1	star	3	none	1	128	4	1	1	8	1	image
2	star	2	x	1	8	2	8	1	1	4	image
2	star	2	y	1	128	2	2	1	128	1	image
2	star	2	z	1	128	2	1	1	32	1	image
2	star	3	none	1	16	2	8	1	1	1	image
3	star	2	x	1	8	2	8	1	1	1	image
3	star	2	y	1	128	2	2	2	32	1	image
3	star	2	z	1	16	2	4	1	1	1	image
3	star	3	none	1	16	4	2	1	16	2	image
4	star	2	x	1	64	4	4	1	64	1	image
4	star	2	y	1	16	1	16	1	8	4	image
4	star	2	z	1	128	2	1	1	1	1	image
4	star	3	none	1	256	4	1	1	1	128	image
5	star	2	x	1	16	4	4	1	1	2	image
5	star	2	y	1	16	2	8	1	1	1	image
5	star	2	z	1	128	2	1	1	1	1	image
5	star	3	none	1	16	2	4	1	1	2	image
2	diamond	2	x	1	128	2	2	1	1	16	image
2	diamond	2	y	1	128	2	1	1	64	1	image
2	diamond	2	z	1	128	2	1	1	1	1	image
2	diamond	3	none	1	128	1	2	1	1	1	image
3	diamond	2	x	1	128	4	2	1	64	1	image
3	diamond	2	y	1	64	1	2	2	1	32	local
3	diamond	2	z	1	32	8	1	4	1	1	local
3	diamond	3	none	1	32	2	2	1	1	1	image
4	diamond	2	x	1	8	16	2	1	1	64	local
4	diamond	2	y	1	32	1	8	2	1	32	local
4	diamond	2	z	1	32	8	1	1	4	1	local
4	diamond	3	none	1	64	4	2	1	1	4	image
5	diamond	2	x	1	4	32	2	1	1	64	local
5	diamond	2	y	1	32	1	4	2	1	16	local
5	diamond	2	z	1	32	8	1	2	2	1	local
5	diamond	3	none	1	128	2	1	1	1	8	image
1	no corners	3	none	1	128	2	1	1	1	1	image
2	no corners	2	x	1	64	2	4	1	1	1	image
2	no corners	2	y	1	32	1	8	1	1	1	image
2	no corners	2	z	1	32	4	1	2	2	1	local
2	no corners	3	none	1	32	8	1	1	1	64	local
3	no corners	2	x	1	16	8	2	1	1	64	local
3	no corners	2	y	1	64	1	2	2	1	32	local
3	no corners	2	z	1	32	4	1	1	4	1	local
3	no corners	3	none	1	32	8	2	1	1	32	local
4	no corners	2	x	1	8	16	2	1	1	64	local
4	no corners	2	y	1	64	1	2	1	1	32	local
4	no corners	2	z	1	32	8	1	2	2	1	local
4	no corners	3	none	1	64	8	2	1	1	64	local
5	no corners	2	x	1	8	8	4	1	2	32	local
5	no corners	2	y	1	64	1	2	1	1	16	local
5	no corners	2	z	1	32	8	1	1	4	1	local
5	no corners	3	none	1	64	4	1	1	1	256	local

Table 13

Radius	Type	Dims	Unique Dim	BX/VX	WX	WY	WZ	CX	CY	CZ	Data loading technique
1	thumbtack	3	x	4	16	16	1	1	1	1	vec
1	thumbtack	3	y	1	256	1	1	1	1	2	global
1	thumbtack	3	z	1	64	1	2	2	1	1	global
2	thumbtack	3	x	4	16	4	1	1	1	1	vec
2	thumbtack	3	y	1	32	4	2	1	2	16	local
2	thumbtack	3	z	1	32	4	2	1	2	32	local
3	thumbtack	3	x	4	16	4	2	1	1	1	vec
3	thumbtack	3	y	1	16	8	2	1	2	32	local
3	thumbtack	3	z	1	16	8	2	1	2	32	local
4	thumbtack	3	x	4	16	4	4	1	1	1	vec
4	thumbtack	3	y	1	8	8	4	2	1	32	local
4	thumbtack	3	z	1	8	8	4	2	1	32	local
5	thumbtack	3	x	1	16	8	2	1	2	64	local
5	thumbtack	3	y	1	8	16	2	2	1	64	local
5	thumbtack	3	z	1	64	2	1	1	2	1	global
0	dense	1	none	1	128	1	1	2	1	4	global
1	dense	1	x	1	128	2	1	1	1	2	global
1	dense	1	y	2	32	8	1	1	1	1	vec
1	dense	1	z	1	64	1	4	1	1	8	global
1	dense	2	x	4	16	8	1	1	1	1	vec
1	dense	2	y	1	64	1	1	2	1	1	global
1	dense	2	z	1	64	2	1	2	1	1	global
1	dense	3	none	1	32	4	2	1	4	16	local
2	dense	1	x	1	128	2	1	2	1	2	global
2	dense	1	y	2	32	2	4	1	1	2	vec
2	dense	1	z	2	32	1	8	1	2	1	vec
2	dense	2	x	4	8	8	2	1	1	32	vec
2	dense	2	y	1	64	1	4	4	1	8	local
2	dense	2	z	1	32	8	1	2	4	1	local
2	dense	3	none	1	32	4	2	1	2	8	local
3	dense	1	x	1	64	1	1	4	1	1	global
3	dense	1	y	4	16	4	1	1	1	2	vec
3	dense	1	z	4	32	1	8	1	2	1	vec
3	dense	2	x	4	16	4	4	1	1	1	vec
3	dense	2	y	1	128	1	2	2	1	32	local
3	dense	2	z	1	32	8	1	4	2	1	local
3	dense	3	none	1	32	4	2	1	1	16	local
4	dense	1	x	1	64	2	1	2	1	1	global
4	dense	1	y	4	32	2	1	1	1	1	vec
4	dense	1	z	4	16	2	8	1	1	2	vec
4	dense	2	x	4	16	4	4	2	1	4	vec
4	dense	2	y	1	32	1	8	4	1	1	local
4	dense	2	z	1	32	8	1	2	4	1	local
4	dense	3	none	1	2	16	8	1	1	16	local
5	dense	1	x	1	128	2	1	2	1	1	global
5	dense	1	y	4	16	16	1	1	1	4	vec
5	dense	1	z	4	32	1	8	1	1	2	vec
5	dense	2	x	4	16	16	1	1	16	1	vec
5	dense	2	y	1	128	1	2	2	1	16	local
5	dense	2	z	1	32	8	1	2	4	1	local
5	dense	3	none	1	32	2	4	1	1	64	local
1	star	2	x	2	64	1	4	1	1	1	vec
1	star	2	y	1	128	1	1	2	1	1	global

Table 13: Continued.

Radius	Type	Dims	Unique Dim	BX/VX	WX	WY	WZ	CX	CY	CZ	Data loading technique
1	star	2	z	1	64	1	2	4	1	1	global
1	star	3	none	1	64	1	1	4	1	1	global
2	star	2	x	4	16	4	4	1	1	2	vec
2	star	2	y	1	128	1	2	1	1	4	global
2	star	2	z	1	64	4	1	2	1	1	global
2	star	3	none	1	64	1	2	4	1	1	global
3	star	2	x	4	8	4	8	2	1	2	vec
3	star	2	y	1	64	1	4	1	1	16	global
3	star	2	z	1	32	8	1	4	2	2	local
3	star	3	none	1	64	1	4	1	2	1	global
4	star	2	x	4	8	4	8	2	1	1	vec
4	star	2	y	1	64	1	4	2	1	1	global
4	star	2	z	1	32	8	1	2	4	1	local
4	star	3	none	1	64	1	4	1	1	8	global
5	star	2	x	4	16	4	4	1	1	2	vec
5	star	2	y	1	128	1	2	2	1	32	local
5	star	2	z	1	32	8	1	1	8	1	local
5	star	3	none	1	64	1	2	1	1	8	global
2	diamond	2	x	4	16	4	4	1	1	4	vec
2	diamond	2	y	1	64	1	4	1	1	4	global
2	diamond	2	z	1	64	2	2	2	1	1	global
2	diamond	3	none	1	32	4	2	1	2	32	local
3	diamond	2	x	4	16	8	2	1	32	1	vec
3	diamond	2	y	1	128	1	2	2	1	32	local
3	diamond	2	z	1	16	16	1	4	2	1	local
3	diamond	3	none	1	64	4	1	1	32	1	global
4	diamond	2	x	4	16	4	4	1	1	1	vec
4	diamond	2	y	1	32	1	8	4	1	16	local
4	diamond	2	z	1	32	8	1	2	4	1	local
4	diamond	3	none	1	8	8	4	2	1	32	local
5	diamond	2	x	4	16	8	2	1	16	1	vec
5	diamond	2	y	1	128	1	2	2	1	32	local
5	diamond	2	z	1	32	8	1	1	8	1	local
5	diamond	3	none	1	16	4	4	1	1	32	local
1	no_corners	3	none	1	64	1	1	2	1	1	global
2	no_corners	2	x	4	16	4	4	1	1	16	vec
2	no_corners	2	y	1	64	1	4	4	1	16	local
2	no_corners	2	z	1	32	8	1	2	4	1	local
2	no_corners	3	none	1	32	4	2	1	2	8	local
3	no_corners	2	x	4	16	4	4	2	1	1	vec
3	no_corners	2	y	1	128	1	2	2	1	32	local
3	no_corners	2	z	1	32	8	1	4	2	1	local
3	no_corners	3	none	1	32	4	2	1	1	16	local
4	no_corners	2	x	4	16	8	2	1	1	1	vec
4	no_corners	2	y	1	32	1	8	4	1	1	local
4	no_corners	2	z	1	32	8	1	2	4	1	local
4	no_corners	3	none	1	2	16	8	1	1	16	local
5	no_corners	2	x	4	16	16	1	1	16	1	vec
5	no_corners	2	y	1	128	1	2	2	1	16	local
5	no_corners	2	z	1	32	8	1	2	4	1	local
5	no_corners	3	none	1	32	2	4	1	1	64	local

Conflicts of Interest

There are no conflicts of interest related to this paper.

Acknowledgments

This work was supported by grants from The Natural Sciences and Engineering Research Council of Canada (NSERC) and from Qualcomm Canada.

References

[1] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2013*, pp. 519–530, usa, June 2013.

[2] D. Lowell, J. Godwin, J. Holewinski et al., "Stencil-aware GPU optimization of iterative solvers," *SIAM Journal on Scientific Computing*, vol. 35, no. 5, pp. S209–S228, 2013.

[3] M. J. Gibson, E. C. Keedwell, and D. A. Savić, "An investigation of the efficient implementation of cellular automata on multi-core CPU and GPU hardware," *Journal of Parallel and Distributed Computing*, vol. 77, pp. 11–25, 2015.

[4] NVIDIA Corporation, "OpenCL programming guide for the CUDA architecture," 2010.

[5] A. Ganapathi, K. Datta, A. Fox, and D. Patterson, "A case for machine learning to optimize multicore performance," in *Proceedings of the 1st USENIX Conference on Hot Topics in Parallelism (HotPar '09)*, Berkeley, Calif, USA, 2009.

[6] L. Breiman, "Random forest," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001.

[7] NVIDIA Corporation, *NVIDIA's next generation CUDA compute architecture: Kepler GK110*, 2012.

[8] AMD, "Amd graphics core next GCN architecture," 2012.

[9] The Khronos Group, "Open Computing Language (OpenCL)," http://www.khronos.org/opencl/.

[10] Y. Yang, P. Xiang, J. Kong, and H. Zhou, "A GPGPU compiler for memory optimization and parallelism management," in *Proceedings of the the 2010 ACM SIGPLAN conference*, p. 86, Toronto, Ontario, Canada, June 2010.

[11] D. Young, "Iterative methods for solving partial difference equations of elliptic type," *Transactions of the American Mathematical Society*, vol. 76, pp. 92–111, 1954.

[12] T. Grosser, A. Cohen, P. H. Kelly, J. Ramanujam, P. Sadayappan, and S. Verdoolaege, "Split tiling for GPUs," in *Proceedings of the the 6th Workshop*, pp. 24–31, Houston, Texas, March 2013.

[13] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs," in *Proceedings of the 23rd International Conference on Supercomputing, ICS'09*, pp. 256–265, usa, June 2009.

[14] T. Han and T. Abdelrahman, "Automatic tuning of local memory use on GPGPUs," in *Proceedings of the ADAPT*, 2015.

[15] S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach," in *1em plus 0.5em minus 0.4em Pearson Education*, Artificial Intelligence, A Modern Approach, 2nd edition, 2003.

[16] A. Chiu, J. Garvey, and T. S. Abdelrahman, "Genesis: a language for generating synthetic training programs for machine learning," in *Proceedings of the 12th ACM International Conference on Computing Frontiers (CF '15)*, Ischia, Italy, May 2015.

[17] OpenCV, "The OpenCV library," http://docs.opencv.org.

[18] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA data mining software: an update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[19] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, 1986.

[20] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on GPU architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing, ICS'12*, pp. 311–320, Italy, June 2012.

[21] K. Datta, M. Murphy, V. Volkov et al., "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '08)*, pp. 1–12, IEEE, Austin, Tex, USA, November 2008.

[22] S. Kamil, C. Chan, S. Williams et al., "A Generalized Framework for Auto-tuning Stencil Computations," in *Proceedings of the Cray User Group Conference*, 2009.

[23] S. Kamil, C. Chan, L. Oliker, J. Shall, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS '10)*, pp. 1–12, IEEE, Atlanta, Ga, USA, April 2010.

[24] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris, "Autotuning stencil-based computations on GPUs," in *Proceedings of the 2012 IEEE International Conference on Cluster Computing, CLUSTER 2012*, pp. 266–274, China, September 2012.

[25] Y. Zhang and F. Mueller, "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters," in *Proceedings of the 10th International Symposium on Code Generation and Optimization, CGO 2012*, pp. 155–164, USA, April 2012.

[26] W. T. Tang, W. J. Tan, R. Krishnamoorthy et al., "Optimizing and auto-tuning iterative stencil loops for gpus with the in-plane method," in *Proceedings of the 2013 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 452–462, Cambridge, Mass, USA, May 2013.

[27] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive auto-tuning with boosted regression trees," in *Proceedings of the 2012 Innovative Parallel Computing (InPar '12)*, May 2012.

[28] A. Magni, D. Grewe, and N. Johnson, "Input-aware auto-tuning for directive-based GPU programming," in *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units, GPGPU 2013*, pp. 66–75, March 2013.

[29] W. Jia, K. A. Shaw, and M. Martonosi, "Starchart: Hardware and software optimization using recursive partitioning regression trees," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT 2013*, pp. 257–267, September 2013.

[30] J. Ansel, S. Kamil, K. Veeramachaneni et al., "OpenTuner," in *Proceedings of the the 23rd international conference*, pp. 303–316, August 2014.

[31] M. Christen, O. Schenk, and H. Burkhart, "PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*, pp. 676–687, IEEE, Anchorage, Alaska, USA, May 2011.

[32] OpenTuner, "OpenTuner Project Git," https://github.com/jansel/opentuner/issues/87.

[33] W. Feng and T. S. Abdelrahman, "A sampling based strategy to automatic performance tuning of GPU programs," in *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium: Workshops (IPDPSW)*, pp. 1342–1349, Lake Buena Vista, Fla, USA, May 2017.

[34] P. S. Rawat, C. Hong, M. Ravishankar et al., "Resource Conscious Reuse-Driven Tiling for GPUs," in *Proceedings of the the 2016 International Conference*, pp. 99–111, Haifa, Israel, September 2016.

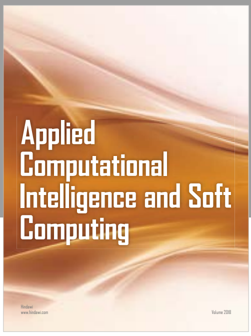
[35] J. D. Garvey and T. S. Abdelrahman, "Automatic Performance Tuning of Stencil Computations on GPUs," in *Proceedings of the 2015 44th International Conference on Parallel Processing (ICPP)*, pp. 300–309, Beijing, China, September 2015.

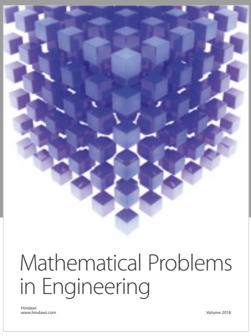
[36] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proceedings of the 2012 Innovative Parallel Computing (InPar '12)*, May 2012.

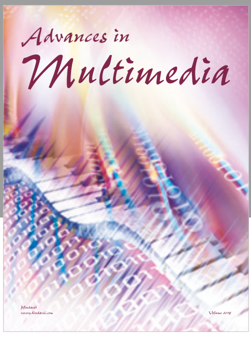
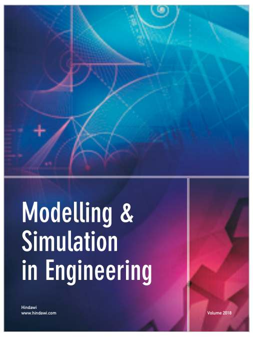
[37] J. W. Choi, A. Singh, and R. W. Vuduc, "Model-driven autotuning of sparse matrix-vector multiply on GPUs," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, pp. 115–126, ACM, Bangalore, India, January 2010.

[38] Y. Li, J. Dongarra, and S. Tomov, "A note on auto-tuning GEMM for GPUs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 5544, no. 1, pp. 884–892, 2009.

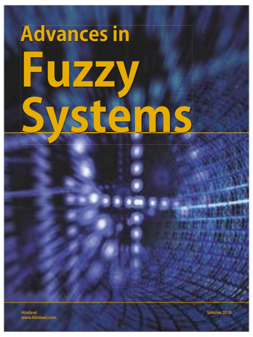
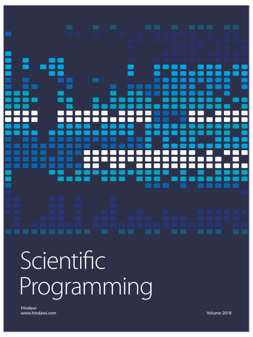
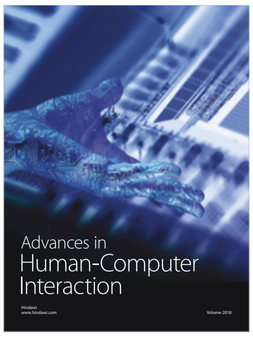
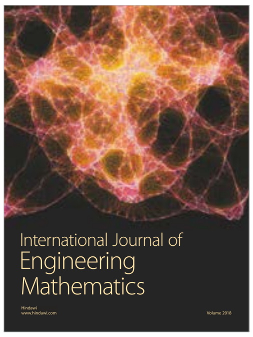
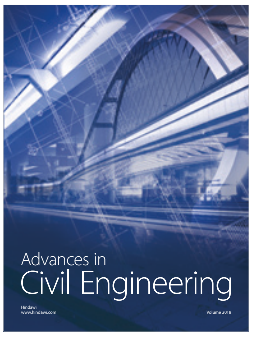
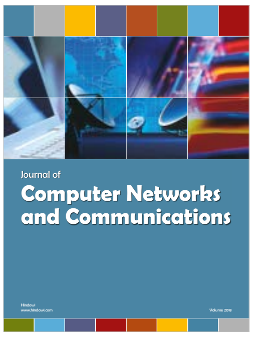
[39] A. Nukada and S. Matsuoka, "Auto-tuning 3-D FFT library for CUDA GPUs," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*, November 2009.


Applied Computational Intelligence and Soft Computing

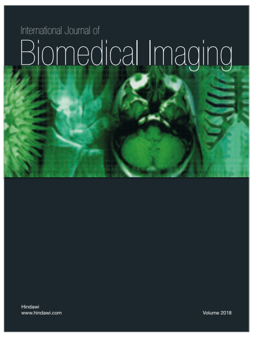
The Scientific World Journal

Mathematical Problems in Engineering

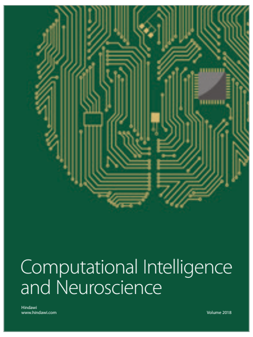
Journal of Engineering

Advances in Multimedia

Modelling & Simulation in Engineering

Advances in Artificial Intelligence

International Journal of Reconfigurable Computing

Advances in Fuzzy Systems

Scientific Programming

Advances in Human-Computer Interaction

International Journal of Engineering Mathematics

Advances in Civil Engineering

Journal of Computer Networks and Communications

Journal of Robotics

International Journal of Computer Games Technology

International Journal of Biomedical Imaging

Journal of Electrical and Computer Engineering

Computational Intelligence and Neuroscience