Automated GPU Grid Geometry Selection for OpenMP Kernels

Taylor Lloyd
University of Alberta
Edmonton, Canada
tjlloyd@ualberta.ca

Artem Chikin
University of Alberta
Edmonton, Canada
artem@ualberta.ca

Sanket Kedia
IIT Kharagpur
Kharagpur, India
kedia.sanket@cse.iitkgp.ernet.in

Dhruv Jain
IIT Kharagpur
Kharagpur, India
dhruvjaincse@iitkgp.ac.in

José Nelson Amaral
University of Alberta
Edmonton, Canada
jamaral@ualberta.ca

***Abstract*—Modern supercomputers are increasingly using GPUs to improve performance per watt. Generating GPU code for target regions in OpenMP 4.0, or later versions, requires the selection of grid geometry to execute the GPU kernel. Existing industrial-strength compilers use a simple heuristic with arbitrary numbers that are constant for all kernels. After characterizing the relationship between region features, grid geometry and performance, we built a machine-learning model that successfully predicts a suitable geometry for such kernels and results in a performance improvement with a geometric mean of 5% across the benchmarks studied. However, this prediction is impractical because the overhead of the predictor is too high. A careful study of the results of the predictor allowed for the development of a practical low-overhead heuristic that resulted in a performance improvement of up to 7 times with a geometric mean of 25.9%. This paper describes the methodology to build the machine-learning model, and the practical low-overhead heuristic that can be used in industry-strong compilers.**

***Index Terms*—GPUs, heterogeneous computing, OpenMP, Machine Learning, Grid Geometry**

I. Introduction

GPUs are composed of tens to hundreds of streaming multiprocessors (SMs), each capable of executing thousands of threads in parallel. A shared program *kernel* is executed by many threads at once, in a data-parallel fashion. When executing a kernel on a GPU, threads are grouped into thread blocks. All threads within a thread-block execute on a single SM, and are therefore able to perform cooperative tasks and share intermediate results. By contrast, threads in different thread-blocks cannot communicate directly. The number of thread blocks and the number of threads per block are collectively referred to as a *grid geometry* and both must be specified when initiating a kernel execution. Languages such as Open Multi-Processing (OpenMP) abstract details of the parallelism model and leave the mapping of parallel constructs up to compilers and dynamic runtimes [4]. The goal of determining an efficient mapping of parallelism to hardware in a portable fashion is well-studied [9, 13]. Adding a new dimension to the task, the compiler/runtime must determine the best grid geometry for each OpenMP `target` region. A `target` region typically consists of parallel loops; existing production runtimes, such as the one used by Clang/LLVM 4.x-capable compiler [1], use a simple heuristic to select the grid geometry for each region.[1]

[1]This simple heuristic is also used by a commercial industrial-strength compiler.

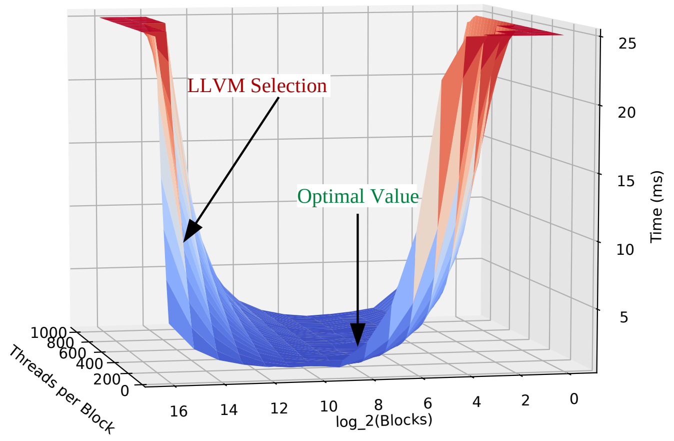

Fig. 1: Execution time of kernel 21, as a function of the number and size of blocks. The best discovered point is 9.8x faster than the LLVM selection.

For Nvidia GPUs, the runtime sets the total number of threads to the number of iterations of the parallel loop, and the number of threads per block, arbitrarily, to 128. This simple selection, which works reasonably well but can result in large queues of thread-blocks, is referred to as the *LLVM selection* in this paper. This paper demonstrates that the compiler's current strategy produces geometries that achieve reasonable performance for OpenMP code that is well-structured for GPU execution. However, the LLVM selection fails to extract performance from OpenMP kernels that exhibit amounts of parallelism that do not optimally map to GPU execution. Such kernels are common and represent a typical case of a CPU programmer converting existing OpenMP code to use accelerator offloading. A good compiler **must be able to produce efficient code even when the program is poorly written**.

The simple, sometimes inefficient, selection mechanism in two industrial-strength compilers indicates that the task of determining GPU grid geometry for OpenMP `target` regions is a problem worth investigating. An extensive empirical study of the relationship between grid geometry and performance for the set of all compilable C/C++ OpenMP benchmarks in the SPEC ACCEL [6] and Unibench [10] benchmark suites revealed that the performance of an individual benchmark can be improved by up to 9.8 times, as shown in Figure 1 with a possible geometric mean of 36% across all the benchmarks studied.

This problem is intuitively well-suited for a predictive-modelling approach where a collection of static and dynamic features are extracted from the `target` region, and a predictor is constructed to output the grid geometry that results in good performance. A random decision forest model successfully predicted the grid geometry and delivered a geometric mean speedup of 5% over to the LLVM selection. However, a closer examination of this predictor **revealed that it is not practical** because it must extract features at program runtime, immediately before kernel launch. The issue is that the time needed to evaluate the features and run the predictor negates any improvement achieved by the use of more efficient grid geometry. [2]

A careful inspection of the random-forest predictor decisions revealed that it exposed choices that can be made analytically. Studying the grid geometry space and the model's predictions, we discovered an effective and inexpensive heuristic that can be used in an industrial-strength compiler to select grid geometry for GPU-intended `target` regions. This new heuristic has the virtue of being simple, **but its discovery had eluded several experienced compiler designers** both in industry and in the open-source compiler community.

This new low-overhead heuristic led to speedups of up to $7\times$ over the LLVM selection, with a geometric mean for the performance improvement of 25.9% for the entire set of benchmarks across SPEC ACCEL and Unibench.

This paper makes the following contributions:

1) An exhaustive characterization across the space of possible grid geometries on a variety of benchmarks in order to understand its effects on performance.
2) An analysis of the efficacy of existing heuristics in the OpenMP 4.x implementation for LLVM/Clang, by comparing to an exhaustive search over possible grid geometries.
3) Description of a methodology to apply machine learning to the problem of grid geometry selection. The result is substantially improved kernel execution time, but impractical prediction overhead.
4) A low-overhead heuristic suitable for production compilers with superior performance compared to the best machine-learning prediction model investigated.

II. Data Collection

To understand how GPU grid geometry affects the performance of OpenMP 4.x programs, we gather performance data at various grid geometry configurations. We evaluate the compiler's current heuristic, and analyze the data for trends that may lead to a better approach. Performance data can also be combined with features extracted statically and dynamically from kernels to produce a dataset for training machine learning models.

[2]When assessing the use of learning approaches in code generation, it is important to ensure that the run-time prediction overhead was taken into account in the reporting of results because sometimes it is overlooked.

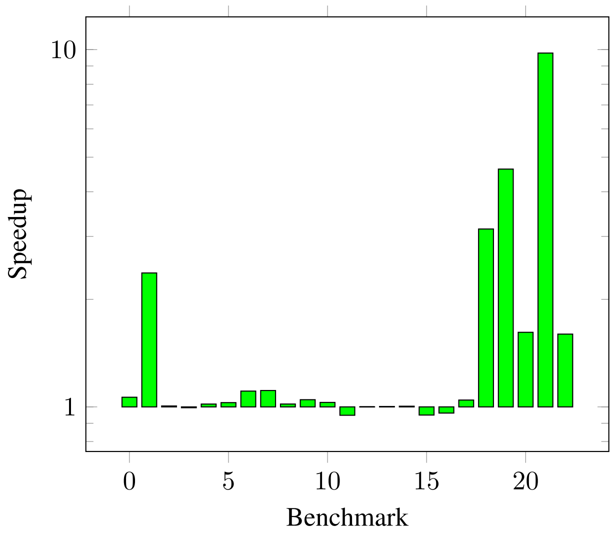

Fig. 2: Improvement available with the best discovered grid geometry versus the LLVM selection.

We examine 23 kernels taken from 11 OpenMP C/C++ benchmarks. Benchmark applications are taken from the SPEC ACCEL [6] and Unibench [10] benchmark suites. Data collection was conducted on a workstation machine with an Intel i7-4770 CPU with 32GB of RAM, running CentOS 6.7. To collect kernel execution times, we use the CUDA 8 drivers and runtime and a Nvidia Titan X Pascal with a locked clock frequency. Reported times are averaged over 5 executions. On each run, for each of the possible grid geometry configurations, all benchmarks are executed in an interleaved fashion, before switching to the next grid configuration. The grid geometry space can be represented as a (t, b) tuple, where t is the number of threads per block and b is the number of blocks. Due to architectural constraints, t can vary between 1 and 1024 threads, while b can vary between 1 and 2^{16} blocks. These ranges yield $2^{16} \times 2^{10} = 2^{26}$ possible combinations, which are far too many to actually execute. However, because warps are executed simultaneously, it makes little sense to execute with a number of threads that does not perfectly fill a warp. Therefore, the search space can be limited to $2^{16} \times 2^{5} = 2^{21}$ combinations, which is still too large. We test only block counts in powers of 2, yielding $17 \times 32 = 544$ total combinations per benchmark. This approach makes a trade-off: potentially missing the true optimal grid geometry in between the test points to complete the search in a reasonable amount of time.

The last challenge for data collection was the sheer amount of time required. Locating near-optimal geometry involves spending many cycles executing benchmarks using extremely poor configurations, resulting in each data collection taking more than a week of compute time.

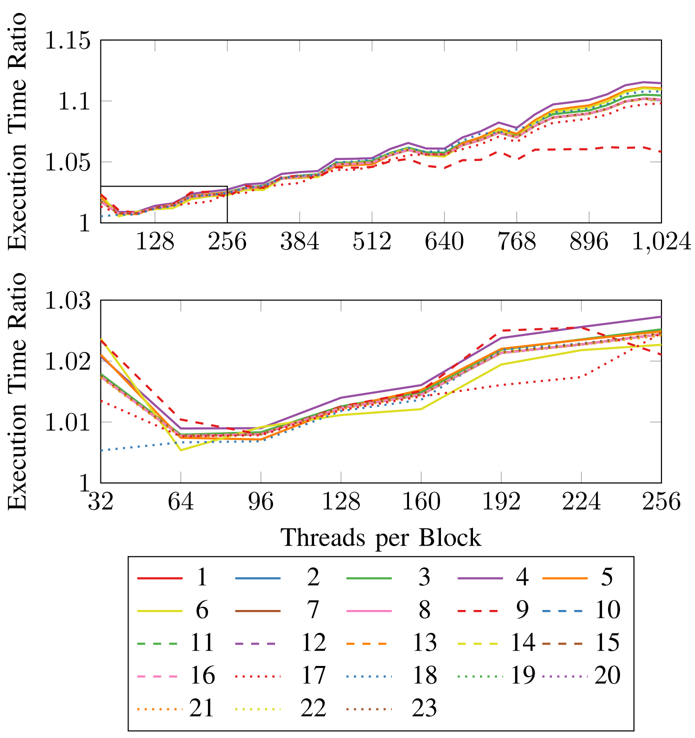

Fig. 3: Minimum Execution Time at set threads per block / Minimum Execution Time. Each line represents 22 of the 23 benchmarks in a leave-one-out manner.

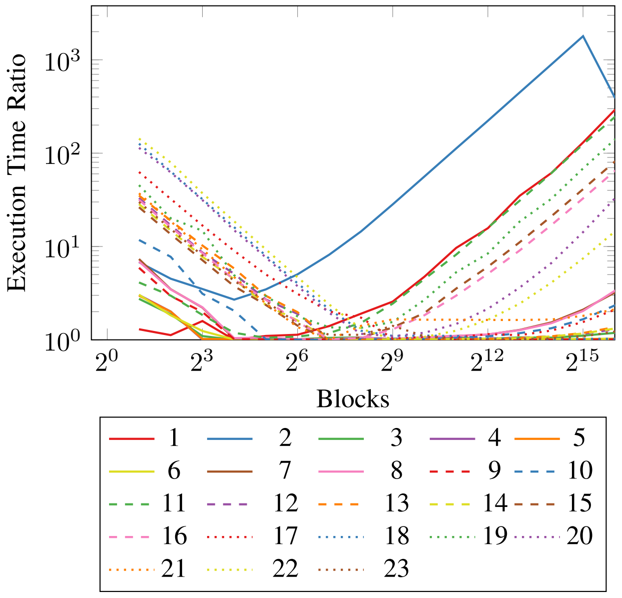

Fig. 4: Minimum Execution Time given 96 threads per block / Minimum Execution Time. Average is weighted by the minimum execution time of each benchmark.

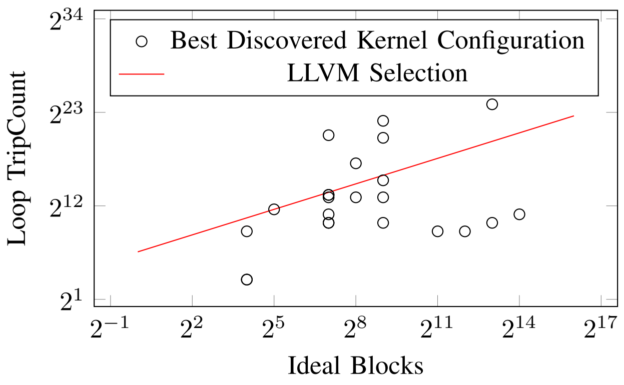

Fig. 5: Number of blocks minimizing execution time given 96 threads per block / Loop Tripcount for each benchmark.

A. Best Discovered Grid Geometry Performance Relative to Compiler Default

An analysis of the collected data allows for a comparison with the simple heuristic currently used to determine the limits in the potential speedup that a change to this heuristic could yield. Figure 2 shows the percentage improvement for each benchmark, over the LLVM selection, when the best discovered grid geometry is used. Speedups are presented on a log scale, to present equivalent speedups and slowdowns as equally-sized bars. Choosing a better grid geometry can yield up to 9.8x improvement, with a geomean of 36% potential improvement across all tested benchmarks. The negative improvement seen in a few of the benchmarks is an artifact of the coarse-grained search used to reduce the data collection time: the course-grain search simply did not test a grid geometry that was as performant as the one currently chosen by the compiler. In these cases the performance difference is marginal (within $\approx 5\%$).

B. Threads Per Block

A relevant question is: *Is there an optimal number of threads per block?* A way to answer this question is to plot the execution time for a range of threads-per-block values for the programs available for the study and then to select the best value. However, this approach has the drawback that it would be using *the same* set of programs both to set a parameter and then to measure performance. To avoid this methodological flaw, the exploration for the best thread-per-block parameter uses a leave-one-out strategy: each fold is formed by 22 of the 23 benchmarks. The execution time for each program for all combinations of number of blocks and threads-per-block for each benchmark is determined by exploration. Figure 3 plots for each fold (identified by the benchmark left out in the legend) the ratio between the sum of the minimum execution time for the benchmarks in the fold for a given threads-per-block value and the sum of the minimum overall execution time. Thus Figure 3 plots the overhead of using a given threads-per-block value over the best execution time found by the search. For most benchmarks, the overhead is minimized at either 64 or 96 threads per block, with performance degrading in both directions. Kernel 18 is a notable outlier that requires explanation.

Based on the results from this exploration, the number of threads per block to be used to evaluate benchmark 6 should be 64, for benchmark 18, it should be 32, and for all other benchmarks 96 threads per blocks should be used. These numbers ensure that information from the benchmark used for evaluation is not used to obtain the prediction. However, to obtain a model to find the number of blocks (SectionII-C) a single value for the number of threads per block is used, and the most common one from Figure 3 is 96.

SIMD Operations in Kernel 18: Kernel 18 makes use of the `omp simd` directive, marking a loop to be parallelized using SIMD units within each thread. Current GPUs do not have SIMD units, so compilers currently attempt to emulate SIMD operations. If a target region contains a SIMD clause, the code generator used by Clang currently sets aside a warp of threads to be used as a large SIMD unit. Because of the dedicated warp, Kernel 18 is penalized for running with 32 threads per warp.

C. *Number of Blocks*

Next the model must predict the total number of blocks. Figure 4 shows the overhead of forcing 96 threads per block and varying the block count for each kernel. Each kernel has a distinct minima, showing that some factor that varies by kernel affects the ideal block count.

The heuristic used in LLVM calculates the number of blocks as a linear function of the *loop tripcount* – the number of parallel loop iterations to be executed. To investigate the validity of that assumption, Figure 5 plots, for each benchmark, the number of blocks required to minimize execution time when 96 threads per block are created, versus the loop tripcount. The plot also shows a line for the LLVM heuristic, assuming 96 threads per block instead of the original 128. If the LLVM heuristic's assumptions were optimal, we would expect a straight line from the bottom left to the top right of the plot. While there is indeed a weak linear relationship, there is clearly still some hidden variable, leaving room for a machine-learning model to discover this relationship and improve performance.

III. Modeling with Machine Learning

Multiple applications of automated learning to parallel systems and complier technologies appear in the literature(e.g. [3, 9, 11, 13]). Tuning of compile-time and launch-time kernel parameters of GPGPU code in particular has attracted a lot of research attention in light of GPUs' popularity [5, 7, 12].

Our initial goal is to model the performance of a GPU kernel generated for a `target` region as a function of the selected grid geometry. The method is to use *offline supervised learning* to create a machine learning model that captures static and dynamic kernel characteristics in an attempt to generate a prediction of the optimal grid geometry. The model uses the dataset acquired through the exhaustive exploration of the grid geometry space for each kernel to train the predictor and is evaluated using leave-one-out cross-validation. To the best of our knowledge, our work is the first to investigate automatic selection of Grid Geometry in the context of OpenMP GPU kernels.

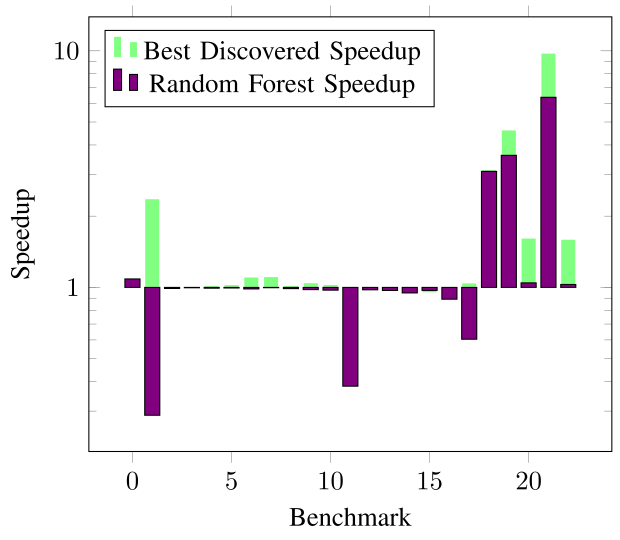

Fig. 6: Speedup over the LLVM selection for the Random Forest Model Predictor grid configuration not including the prediction overhead. This performance cannot be realized in practice. Results are shown on a log scale to present equivalent speedups and slowdowns as equivalently-sized bars.

A. *Finding Additional Features*

Given only a weak linear relationship between the loop tripcount, used by the LLVM selection, and the ideal number of blocks (described in section II-C), we investigate the use of additional features which may impact grid geometry. We introduce simple static analysis techniques to generate additional features, with the goal of further characterizing such kernels. The following features either capture a hardware resource that limits potential occupancy (known at compile-time) or certain code characteristics that would affect GPU utilization.

Stack Frame Size: A sufficiently large stack frame might mean that only one thread block at a time may be scheduled for execution on a given Streaming Multiprocessor (SM) .

Register Count: The per-thread register count affects how many blocks can be scheduled to run on a single SM due to a fixed register file size.

Directive and Clause Use: Separating kernels based on which OpenMP constructs they employ is a way to classify different behavior. We restrict this feature to count only the clauses that may affect inter-thread cooperation within thread-blocks.

Code Size Estimate: We built a simple static analysis approximating the number of instructions executed per loop iteration to capture the amount of work performed by a given thread.

Total Work Estimate: This feature estimates the overall amount of work to be done by a given kernel (size estimate $\times$ tripcount) .

B. *Random Forest Model*

Attempts to build a linear-regression model were not successful. Therefore, we turn to ensemble approaches. Random

Forests are an ensemble learning method that can be used for both classification and regression. They are a combination of tree predictors such that each tree depends on the values of a random vector sampled independently and with the same distribution for all trees in the forest. In this methodology, the model is designed to predict the execution time of a kernel based on a given grid geometry and the features already used for the linear regression model. For each kernel, execution times were obtained for all 544 thread-block combinations. Thus there is a total of $22 * 544 = 11968$ training data points and 544 test points (corresponding to each kernel not used for training). The implementation used is based on Breiman and Cutler's Random Forests for Classification and Regression [2].

At prediction time, the model is queried for the predicted execution time at every thread and block combination measured in the data exploration. The threads per block and block count that correspond to the shortest expected execution time are then selected for kernel launch.

C. Machine Learning Predictor Performance

Figure 6 shows the speedup over the LLVM selection for all the 23 kernels when using the best predicted grid geometry by this random forest model. The graph also displays the optimum speedup achievable through grid geometry, as shown in Figure 2. For data shown in this graph, the predictor is used to obtain both number of threads and number of threads per block that will be used for kernel launch. The performance obtained with these predictions ranges from $3.5\times$ slower to 6.4 times faster, with a geomean speedup of 5% across all benchmarks. This result likely indicates that the model was able to discover relationships between program features that correlate with the kernel's performance.

IV. Production Heuristic

The random forest model successfully predicted the grid geometry and outperformed the existing compiler heuristic, substantially for some benchmarks. However, the time taken to perform that prediction at runtime dwarfed the actual execution time of most benchmarks. With the small workloads used for the grid-geometry space exploration, prediction time would exceed the runtime of many benchmarks.

Having constructed a model that could improve performance, but with impractical overhead, we first attempt to create simpler models that could replicate the prediction accuracy but with lower overhead. The accuracy of a new linear model was too low. Either the relations are truly non linear or the feature set, combined with a limited number of benchmark kernels, was insufficient to build a reasonably accurate linear relationship between the grid configuration parameters and execution time.

Next we examined the dataset, with the goal of figuring out the insights the random forest model had derived. Inspection revealed some intriguing relationships. A loop iteration is the smallest parallelizable unit of work. However, the product of threads per block and blocks that minimizes execution time is often *larger* than the loop trip-count. Thus, counter-intuitively, some threads must be assigned *zero* work in this situation. For some benchmarks, the time is minimized at the first exploration point where all threads are assigned work, but for others, execution time is minimized when there are many more threads. Finally, when the loop trip count becomes too large, this relationship breaks down, and fewer threads/loop iteration are required.

```
GetGeometry ( device, kernel ):
    if kernel.Parallelism ≤ device.SMCount then
        /* Short-Loop Kernels */
        threads = 1;
        blocks = kernel.Parallelism;
    else
        threads = device.ThreadsPerBlock;
        threadLimit = device.ThreadLimit /
          device.ThreadsPerBlock;
        regLimit = device.RegisterLimit /
          (kernel.Registers * device.ThreadsPerBlock);
        sharedLimit = device.SharedMemLimit /
          kernel.SharedMem;
        blocksPerSM = min(threadLimit, regLimit,
          sharedLimit, device.BlockLimit);
        maxBlocks = blocksPerSM * device.SMCount;
        kernelBlocks = kernel.Parallelism / threads;
        if kernelBlocks ≥ maxBlocks then
            /* Long-Loop Kernels */
            blocks = maxBlocks;
        else
            /* Ideal-Loop kernels */
            blocks = kernelBlocks;
        end
    end
    return (threads, blocks)
```

Fig. 7: Final Heuristic Algorithm

When execution time is minimized by using more threads than loop iterations, at least some warps are partially empty, because OpenMP loop iterations are first assigned to blocks, and then to threads within a block. These partially empty warps are less affected by traditional GPU performance problems such as branch divergence and non-coalesced memory accesses [14]. However, using the GPU in this manner is extremely inefficient — the speedup over correct resource utilization is only $\approx 1\%$, which is negligible.

Based on these insights, the kernels studied can be divided into three classes, to be handled separately by a new compiler heuristic:

1) **Short-Loop Kernels** - Kernels that use a small amount of parallelism are likely not well suited for GPU execution. Such kernels tend to be naively translated from parallel code written for CPUs and do not consider the unique characteristics of the GPU architecture. The heuristic can improve the overhead of such benchmarks

by creating blocks of 1 thread each, and distributing work across SMs to treat the GPU more similarly to a large multi-core system. Because blocks of size 1 use so few resources on each SM, they don't suffer as much from the resource-wasting problem described earlier.

These kernels are detected when the loop trip count is less than or equal to the number of available SMs. For these kernels we use 1 thread per block and a number of blocks equal to the loop trip count. When detected by the runtime, the OpenMP specification dictates that offloading must be performed if an accelerator is present in the system; thus, preventing the runtime from making a choice to forego GPU execution of programs ill-suited to it. In the future, the OpenMP 5 standard will allow for the compiler/runtime to implement such computational device selection strategies with the new `concurrent` directive.

2) **Ideal-Loop Kernels** - Kernels that can use an appreciable fraction of the GPU are already well-handled by the existing heuristic. No substantial performance gains can be found here because the grid selected is already relatively optimal.

   These kernels have a loop trip count that is larger than the number of SMs available on the GPU (28 for our Nvidia Titan X Pascal). For these kernels, the heuristic prescribes 96 threads per block, and the number of blocks is $\left\lceil \frac{\text{tripcount}}{96} \right\rceil$ blocks.

3) **Long-Loop Kernels** - Kernels with loop trip counts vastly higher than the GPU can execute simultaneously generate massive queues of blocks, preventing opportunistic work. By limiting the number of blocks executed to the maximum executable by the device, the queuing overhead can be reduced substantially.

   These kernels can be identified at runtime by inspecting both the maximum number of blocks the GPU can execute for this kernel, and inspecting the loop trip count provided by the kernel. If the loop trip count exceeds the product of threads per block and number of blocks, then a kernel falls into this class. For these kernels, the new heuristic prescribes the use of 96 threads per block, and the setting of the number of blocks to the maximum that can be simultaneously loaded on the device without queuing.

Formalizing the features considered by the heuristic, a GPU device descriptor should specify the following properties:

- **SMCount** The number of streaming multiprocessors available on the device.
- **ThreadLimit** The maximum number of threads an SM can hold simultaneously.
- **RegisterLimit** The maximum number of 32-bit registers an SM can hold.
- **SharedMemLimit** The maximum amount of shared memory available on an SM.
- **BlockLimit** The maximum number of blocks an SM can hold.

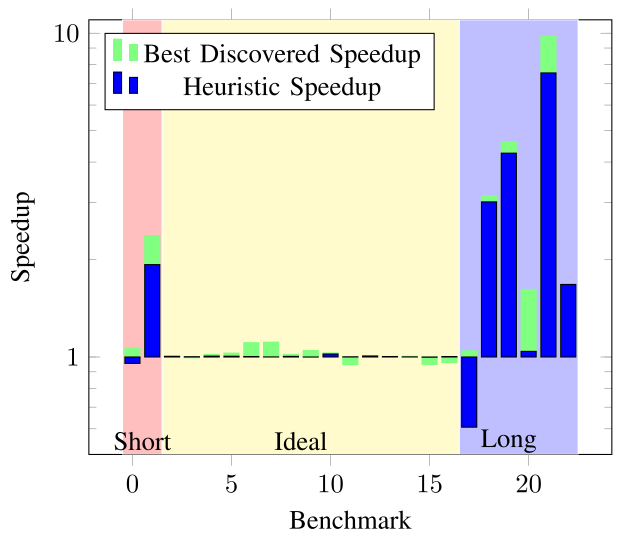

Fig. 8: Speedup for our modified heuristic over the LLVM selection. This performance can be realized in practice. Results are shown on a log scale.

- **ThreadsPerBlock** An experimental value, the ideal threads per block for this device.

Section II-B determined that 96 threads per block is the prediction for most benchmarks. However to use a proper methodology, the evaluation of the heuristic uses the threads-per-block value predicted by the fold that excluded the benchmark that is been evaluated.

The heuristic also requires a kernel descriptor that contains the following properties:

- **Registers** The number of registers required per thread by a kernel.
- **SharedMem** The amount of shared memory required per block by a kernel
- **Parallelism** The number of parallel work units (typically loop iterations) in a kernel

The heuristic pseudocode is shown in Figure 7. The algorithm uses distinct strategies to generate grid geometry for all three kernel classes. It meaningfully captures all of the kernels that we studied and accounts for the behaviours observed by making efficient use of resources. A key insight is to avoid the drastic over provisioning required to truly minimize kernel execution time. An evaluation of this new heuristic against the LLVM selection is shown Figure 8. The kernels on the left have long loops, the kernels on the right have short loops and the ones on the middle have ideal loops. Results range from 39% slower to 7 times faster, with a geomean speedup of 25.9%. The methodology in the evaluation is identical to the one used for data collection presented in Section II.

The speedup comes from the kernels with long or short loops because the kernels with ideal loops are already well-optimized for GPUs by the LLVM selection. Performance improvements are observed in the class of kernels that are poorly written for execution in GPUs, where the heuristic

Benchmark	LLVM	ML Model		Final Heuristic		Best-Discovered Configuration	
	Grid Geometry	Grid Geometry	Speedup	Grid Geometry	Speedup	Grid Geometry	Speedup
S-1	(128,1)	(96,256)	1/1.380	(1,10)	1/1.047	(64,8)	1.064
S-2	(128,1)	(96,1024)	1/23.784	(1,10)	1.930	(32,2)	2.371
S-3	(128,4)	(96,512)	1.005	(64,8)	1.005	(64,4096)	1.006
S-4	(128,4)	(96,512)	1.002	(64,8)	1.002	(64,512)	1/1.005
P-5	(128,4)	(96,2048)	1.004	(64,8)	1.006	(32,256)	1.018
P-6	(128,8)	(96,256)	1.002	(64,16)	1.005	(32,4096)	1.028
P-7	(128,8)	(96,1024)	1/1.030	(64,16)	1.003	(96,128)	1.107
P-8	(128,8)	(96,1024)	1/1.028	(64,16)	1.000	(96,128)	1.111
P-9	(128,8)	(96,1024)	1/1.001	(64,16)	1.004	(32,2048)	1.019
P-10	(128,16)	(96,512)	1/1.009	(64,32)	1/1.000	(96,128)	1.047
P-11	(128,16)	(96,256)	1/1.015	(32,64)	1.022	(512,32)	1.029
S-12	(128,24)	(96,512)	1/2.386	(64,48)	1.001	(32,128)	1/1.055
P-13	(128,64)	(96,512)	1/1.002	(64,128)	1.008	(64,128)	1.001
P-14	(128,64)	(96,1024)	1.002	(64,128)	1.003	(160,128)	1.003
P-15	(128,64)	(96,2048)	1/1.010	(96,256)	1.001	(320,32)	1.004
S-16	(128,79)	(96,512)	1/1.316	(64,156)	1/1.002	(192,64)	1/1.054
S-17	(128,79)	(96,512)	1/1.259	(64,156)	1.003	(160,64)	1/1.041
S-18	(128,256)	(96,512)	1/1.001	(64,448)	1/1.648	(256,128)	1.045
S-19	(128,1024)	(96,256)	3.009	(64,448)	3.014	(128,128)	3.147
P-20	(128,8192)	(96,512)	3.877	(64,448)	4.261	(256,128)	4.630
S-21	(128,10157)	(96,512)	1/1.026	(64,448)	1.041	(96,128)	1.617
P-22	(128,32768)	(96,512)	8.554	(64,448)	7.541	(384,128)	9.779
S-23	(128,125986)	(96,1024)	1.502	(64,448)	1.675	(384,1024)	1.598

TABLE I: Grid Geometry (threads-per-block, blocks) selected for each benchmark by the LLVM selection, our ML model, our proposed heuristic, and exhaustive search. Speedup is shown relative to the LLVM selection. Slowdowns are shown as reciprocals for clarity. Thread-Per-Block values for the the Final Heuristic selected using leave-one-out strategy as described in II-B. S-x are kernels from SPEC ACCEL benchmarks, P-x are kernels from Polybench benchmarks.

causes threads to be separated across SMs. The long-loop kernels generally see large performance improvements, with few slowdowns. The proposed heuristic's goal is to avoid block queuing. Future algorithms may be able to separate and identify cases where block queuing is desirable. The improvement enabled by our algorithm consists of covering a greater variety of possible programs. A singular approach that can both capture performance of the general well-optimized case and edge-cases is of great value: compilers must deal with programs written by expert and non-expert programmers. A good compiler should deliver better performance for all types of programs, whereas existing compilers only did well for coincidentally well-sized programs. Capturing classes of programs with less fitting amounts of parallelism available is especially important because they represent a likely outcome of a naive port of existing CPU-parallel OpenMP code to accelerator offloading with OpenMP 4.x.

While the notion of maximizing device occupancy is a well-established idiom and a default NVIDIA suggestion, existing industrial-strength compiler implementations have neglected to take it into account. That is a strong argument for applying these insights in the context of automatic generation of GPU code from high-level programming models. The low computational complexity makes the runtime overhead of the heuristic negligible and its simplicity allows programmers predictable performance. Since the time of writing, our proposed grid-selection mechanism has been implemented and enabled in a commercially available compiler.

The grids chosen for each kernel, and the associated speedups are shown in Table I.

A. Edge-Case: OpenMP SIMD

Our proposed heuristic matches, or exceeds, the performance of the existing heuristic on 22 of 23 kernels, and is within 10% of the best discovered performance on 19 of 23 kernels. Performance was substantially degraded for one kernel. Benchmark 18 makes use of the `omp simd` construct, which, according to the specification, directs OpenMP implementations to implement the following loop using SIMD vector units. The selection of threads per block, using the leave-one-out strategy described in Section II-B, indicates 32 threads to be the value most likely to maximize performance. SIMD execution on a GPU is emulated using additional dedicated warps of threads, because current GPUs do not have SIMD units. As an artifact of the code-generation scheme, the SIMD region is serialized, leading to poor performance. Until GPUs incorporate SIMD units, code-generation for the `simd` pragma in GPU code will remain a crutch that leads to inefficient code. Still, to maximize performance, the correct strategy is to allocate extra warps to accommodate the SIMD construct code generation. To generalize this insight, more code that utilizes SIMD constructs is needed.

B. Implications of Volta

The Volta architecture introduces several changes that would require minor adjustments to the heuristic approach presented in this paper [8]. Pascal generation cards, which have been used in this work, have SMs that can issue an instruction for 64 threads per cycle. This number gives insight to the discovery made during grid geometry search space exploration that experimentally deemed 96 threads per block to be a reasonable choice for most OpenMP kernels we have encountered. By slightly over prescribing the number of threads per block to the number of threads that can be issued an instruction each cycle, a sufficient amount of latency-hiding can be achieved without suffering the excessive scheduling overhead. In Volta, individual SMs have higher core counts and can issue an instruction to double the number of threads per cycle. While we expect our insights to scale similarly to the new architecture, a new set of experiments, similar to the ones performed in section II, is required to derive the **ThreadsPerBlock** value for Volta.

V. Conclusion

Finding a heuristic that can perform well on a diverse set of programs can be a challenging task that requires extensive analysis. Machine learning has been recently gaining traction as a tool in the compiler researcher's toolbox that can model characteristics of program behavior. Despite strengths in capturing unknown relationships to produce meaningful predictions, using machine learning to model program performance has drawbacks. Collecting sufficient programs to successfully predict performance can be far more difficult than to invent a heuristic that achieves the same result. Moreover, when the model uses features that are only available at runtime, the collection of these features and the execution of the model may incur unacceptable overhead. This paper addresses the problem of tuning the GPU grid geometry for kernels generated from OpenMP 4.x programs that use accelerator offloading constructs. Because prediction must happen at runtime, even a successfully tuned model proved unusable because the predictor time often exceeded kernel execution time. Generation of a superior grid geometry by the machine-learning model yielded useful insights that led to the creation of a very practical heuristic. This hybrid approach of machine learning as a means to inform or guide researchers shows that predictive models can not only be used to directly make decisions, but also to aid the creation of heuristics through expert knowledge.

Acknowledgements

This research was supported by the IBM Canada Software Lab Centre for Advanced Studies (CAS) and by the National Science and Engineering Research Council (NSERC) of Canada through their Collaborative Research and Development (CRD) program and through the MITACS research internship program.

References

[1] C. Bertolli, S. F. Antao, G.-T. Bercea, A. C. Jacob, A. E. Eichenberger, T. Chen, Z. Sura, H. Sung, G. Rokos, D. Appelhans, et al. Integrating GPU support for OpenMP offloading directives into Clang. In *Workshop on the LLVM Compiler Infrastructure in HPC (LLVM)*. ACM, 2015.

[2] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[3] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. PACT '08. ACM, 2008.

[4] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 01 1998.

[5] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, 05 2012.

[6] G. Juckeland, W. Brantley, S. Chandrasekaran, B. Chapman, S. Che, M. Colgrove, H. Feng, A. Grund, R. Henschel, W.-M. W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. Stratton, A. Titov, K. Wang, M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran. *SPEC ACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance*. Springer International Publishing, 2015.

[7] S. Lee and R. Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10. IEEE Computer Society, 2010.

[8] Nvidia. NVIDIA TESLA V100 GPU ARCHITECTURE. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: 2018-01-01.

[9] M. F. P. O'Boyle, Z. Wang, and D. Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Code Generation and Optimization (CGO)*. IEEE, 2013.

[10] D. Rolls, C. Joslin, and S.-B. Scholz. Unibench: a tool for automated and collaborative benchmarking. In *International Conference on Program Comprehension (ICPC)*, pages 50–51. IEEE, 2010.

[11] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle. Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *Programming Language Design and Implementation (PLDI)*. ACM, 2009.

[12] M. Vollmer, B. J. Svensson, E. Holk, and R. R. Newton. Meta-programming and auto-tuning in the search for high performance gpu code. In *Proceedings of the 4th ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC 2015. ACM, 2015.

[13] Z. Wang and M. F. O'Boyle. Mapping parallelism to multicores: A machine learning based approach. In *Principles and Practice of Parallel Programming (PPoPP)*. ACM, 2009.

[14] P. Xiang, Y. Yang, and H. Zhou. Warp-level divergence in GPUs: Characterization, impact, and mitigation. In *High Performance Computer Architecture (HPCA)*, pages 284–295. IEEE, 2014.