Autotuning OpenACC Work Distribution via Direct Search

Calvin Montgomery
ctm0012@auburn.edu

Jeffrey L. Overbey
joverbey@auburn.edu

Xuechao Li
xzl0031@auburn.edu

Department of Computer Science and Software Engineering
Auburn University, AL, USA

ABSTRACT

OpenACC provides a high-productivity API for programming GPUs and similar accelerator devices. One of the last steps in tuning OpenACC programs is selecting values for the num_gangs and vector_length clauses, which control how a parallel workload is distributed to an accelerator's processing units. In this paper, we present OptACC, an autotuner that can assist the programmer in selecting high-quality values for these parameters, and we evaluate the effectiveness of two direct search methods in finding solutions. We assess the quality of the the num_gangs and vector_length values found by our autotuner by comparing them to the values found by a bounded exhaustive search; we also compare the kernel execution times to those of the untuned kernel. On a suite of 36 OpenACC kernels, one or both of our autotuner's direct search methods identified values within the top 5% for 29 of the kernels, within the top 10% for five kernels, and within the top 25% for the remaining two. Eleven of the kernels achieved a speedup greater than 2× over the compiler's defaults, and the autotuner required only 7–11 runs of the target program, on average.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

Keywords

accelerators, autotuning, GPUs, OpenACC, XSEDE

1. INTRODUCTION

The OpenACC application programming interface [3] defines OpenMP-like directives for C/C++ and Fortran that allow loops to be marked for parallelization on a GPU or similar accelerator device.

```
#pragma acc parallel loop
for (i = 0; i < n; ++i)
    sum[i] = a[i] + b[i];
```


XSEDE '15, July 26 - 30, 2015, St. Louis, MO, USA

ISBN 978-1-4503-3720-5/15/07…$15.00

DOI: http://dx.doi.org/10.1145/2792745.2792783

As a simple example, the preceding loop adds the elements of two arrays `a` and `b`, where each of the `n` iterations of the loop is run as a separate thread on the accelerator (GPU).

For XSEDE users, OpenACC support is available via the PGI compilers on Comet. GCC 5 will include OpenACC support,[1] which may eventually allow OpenACC programs to be compiled on other XSEDE resources (e.g., Stampede).

OpenACC is important to the XSEDE community because it is especially well suited to scientific computing. CUDA and OpenCL are the dominant APIs for programming GPUs and other accelerator devices, but OpenACC allows programmers to write GPU code more *quickly and productively*. In the previous example, only a single pragma is needed to offload the *for* loop to a GPU. The details of initializing the GPU, transferring data, decomposing the computation into threads, and launching the GPU kernel are all left to the OpenACC compiler. In CUDA or OpenCL, these responsibilities are all left to the programmer; writing this same kernel takes significantly more code.

However, to achieve good performance, it is often necessary to override some of the compiler's decisions. This paper focuses on two clauses, num_gangs and vector_length, which control how the iterations of a loop are distributed to the accelerator's processing units. For example, appending

```
num_gangs(16), vector_length(32)
```

to the pragma in the previous example causes the PGI compiler to translate it into a CUDA kernel that launches 16 thread blocks with 32 threads per block.

Choosing optimal values for num_gangs and vector_length is difficult, as they are affected by the hardware and compiler as well as characteristics of the kernel itself. A static inspection of the code does not readily lead to values that will produce optimal runtime performance.

This paper makes the following contributions:

- We describe OptACC, an autotuner that can, for a fixed input size, find high-quality values for num_gangs and vector_length. OptACC repeatedly selects candidate values, compiles and runs the target kernel with those values, and evaluates their impact on execution time. It is open source and available to XSEDE users.
- We illustrate the search spaces formed by several kernels and describe empirically successful direct search methods (Nelder-Mead and Coordinate Search).
- We evaluate our autotuner on 36 OpenACC kernels to assess its effectiveness, considering the number of points tested and comparing its results to compiler defaults and the results of a bounded exhaustive search.

[1] See https://gcc.gnu.org/wiki/OpenACC

2. RELATED WORK

Autotuning has primarily been successful in optimizing implementations of particular algorithms; FFTW [11] and SPIRAL [14] are two well-known examples. Fully general autotuning—searching arbitrary configuration spaces without *a priori* knowledge of the application domain—has been less successful and is a much more challenging problem; one recent effort is OpenTuner [5], which provides a framework for building domain-specific autotuners by providing implementations of several general-purpose search algorithms and providing the ability to incorporate domain-specific heuristics and construct ensembles of search techniques.

Generally speaking, autotuning problems can be modeled as mathematical optimization problems. Potential values for the variables to be autotuned are collected in a tuple $\mathbf{t}$. Letting $f(\mathbf{t})$ denote the running time of the program with that particular set of tuning parameters, the goal of the autotuning process is to minimize $f(\mathbf{t})$ subject to constraints on the individual tuning parameters. The function f is called the *objective function*.

Evaluating f at a particular point requires running the program and measuring its execution time. In other words, the objective function is expensive to evaluate. Furthermore, derivatives can only be approximated by finite differences. So-called *derivative-free* optimization methods are designed to find local extrema of such functions; *direct search* methods are derivative-free methods that proceed based solely upon evaluations of the objective function without approximating derivatives or model building [10, p. 115]. In this paper, we focus on two classical direct search methods: the Nelder-Mead method [13] and Coordinate Search [10, p. 116].

Balaprakash, Wild, and Hovland [6] investigated using direct search algorithms to identify sequences of transformations to improve the performance of linear algebra kernels (specifically, applying loop unrolling, scalar replacement, loop parallelization, loop vectorization, and register tiling). They implemented a random search, a simple genetic algorithm, and two variations on the Nelder-Mead direct search method (one of which we adopted in our work).

Siddiqui and Feki [15] addressed a more constrained problem. Given an OpenACC loop nest, they identified loops on which to place *gang* and *vector* clauses, and then chose the number of gangs and a vector length. They used a historic learning approach. In a learning phase, they built a database with the best tuning parameters for a particular application, for different input sizes. To autotune a new input size for the same application, they consulted that application's historic data for a similar input size and used it as a starting point for their search.

Magni, Grewe, and Johnson [8] also addressed the problem of tuning the number of gangs and the vector length in OpenACC loop nests. They concluded that (1) autotuning could result in a speedup of up to 4.8× over compiler defaults, and (2) ideal numbers of gangs and vector lengths vary according to input size, although similar numbers can be used for similar input sizes. In their approach, training data is constructed by performing a random search over many configurations (e.g., 2000). Then, when running the program with a new input size, they used a Nearest Neighbor approach, beginnning with configurations that worked well for a nearby point (in terms of Euclidean distance).

The goal of the present paper is somewhat different. Unlike the aforementioned papers [8, 15], our autotuner is designed to find high-quality values for the number of gangs and vector length for a *fixed* input size, without the benefit of training or historic data.

3. TUNING OPENACC

NVIDIA suggests five steps for porting applications to OpenACC (under the marketing slogan "2x in 5 steps" [2]), which are described in more detail in a *PGInsider* article [9]. For most GPU codes, the most important aspect of tuning is minimizing data transfer between the host and the GPU. NVIDIA's five steps reflect this. However, NVIDIA's fifth and final tuning step ("optimize parallel scheduling") is to add OpenACC clauses to override suboptimal compiler-generated loop schedules. The performance impact is often small, relative to the execution time of the entire application, but it is nevertheless an important tuning step, and one that is difficult to do by hand. The present work focuses on this step.

According to the OpenACC standard [4, p. 8], "OpenACC exposes these three levels of parallelism via gang, worker and vector parallelism. Gang parallelism is coarse-grain. A number of gangs will be launched on the accelerator. Worker parallelism is fine-grain. Each gang will have one or more workers. Vector parallelism is for SIMD or vector operations within a worker."

The OpenACC specification intentionally does not specify how these concepts should be mapped to any particular hardware devices. However, for XSEDE users (and most OpenACC developers at the present time), the only available OpenACC compiler is the PGI compiler, and the accelerator device will be an NVIDIA GPU. Thus, it is helpful to recognize how the PGI compiler maps these OpenACC concepts to NVIDIA hardware.

The NVIDIA GPU architecture is built around an array of multithreaded *streaming multiprocessors (SMs)*, which execute in parallel. Each SM contains several *streaming processors (SPs)*, each of which executes a separate thread.

Typically, a GPU kernel consists of thousands of threads. These threads are grouped into *thread blocks*. Each thread block is assigned to a single SM. The threads within a block are divided into 32-thread groups called *warps*, whose instructions issue in SIMD fashion. Each thread in a warp is executed by a different SP in the SM.

As noted in Section 1, given the OpenACC clauses

```
num_gangs(16), vector_length(32),
```

the PGI compiler generates a CUDA kernel that launches 16 thread blocks with 32 threads per block. In other words, the gangs are divided among the SMs, and the vector length determines the number of CUDA threads in each block.[2]

Finding optimal values for num_gangs and vector_length is difficult. They are affected by the hardware (e.g., cache memory and register pressure), the compiler (e.g., transformations performed, loop scheduling, register allocation, and instruction selection), as well as characteristics of the kernel itself (e.g., input size, ratio of memory accesses to floating point operations, and usage of functional units on the accelerator). Therefore, determining ideal num_gangs and vector_length values statically seems infeasible. Finding such values empirically is the goal of our autotuner.

[2]If a `num_workers` clause is present, the PGI compiler appears to create a two-dimensional thread block, so the total number of CUDA threads in the block is *num_workers* × *vector_length*.

```
$ cat example.c
#include <stdio.h>
#define N 16772240
int main() {
  static double a[N];
  #pragma acc parallel loop num_gangs(NUM_GANGS), vector_length(VECTOR_LENGTH)
  for (int i = 0; i < N; i++)
    a[i] = i;
  return 0;
}
$ ~/OptACC/tuner.py \
    --compile-command 'pgcc -acc -ta=nvidia,time -DNUM_GANGS={num_gangs} -DVECTOR_LENGTH={vector_length} example.c' \
    --executable ./a.out \
    --kernel-timing
INFO [num_gangs: 256, vector_length: 128] Average: 1.051543, Standard Deviation: 0.004175
INFO [num_gangs: 224, vector_length:  64] Average: 1.052142, Standard Deviation: 0.004362
INFO [num_gangs: 224, vector_length: 128] Average: 1.052428, Standard Deviation: 0.004341
INFO [num_gangs: 256, vector_length:  64] Average: 1.051670, Standard Deviation: 0.004209
INFO [num_gangs: 288, vector_length: 128] Average: 1.051687, Standard Deviation: 0.004241
INFO -------------
INFO Tested 5 points
INFO Best result found: num_gangs=256  vector_length=128  => time=1.0515431 (stdev=0.0041747368633)
```

Figure 1: Running the OptACC autotuner on an example program using PGI-generated timing output.

4. AUTOTUNER

We have developed an autotuner, called OptACC, that given an OpenACC parallel construct (typically a parallel loop), identifies values that can be supplied in num_gangs and vector_length clauses to improve execution time. OptACC compiles the program with a particular choice of values, runs the program, and assesses the running time of the kernel. Then, it adjusts the values, recompiles, reruns, and repeats this process until it converges on values that appear to result in the smallest running time. An example run is shown in Figure 1.

The user invoking the autotuner typically supplies

- a **target program**, where the OpenACC parallel region(s) to autotune have been annotated as in Figure 1;
- a **build command**, which the autotuner can run to recompile the target program with a particular choice of values for num_gangs and vector_length; and
- the **executable** generated by the build command.

Then, the autotuner repeats the following steps.

1. **Select candidate values for num_gangs and vector_length.** By default, the autotuner chooses values in the range 2–1024, although the user can override the minimum and maximum values for num_gangs and vector_length.
2. **Compile the target program.** The candidate values for num_gangs and vector_length can be inserted directly into the compile command, as shown in Figure 1 (where they are used in externally-defined macros). They are also stored in environment variables so they can be accessed from a Makefile.
3. **Run the target program, and extract timing information from its output.** (See below for details.)
4. **Terminate, or repeat from Step 1.** Based on the timing results, the autotuner will either terminate (after it has found an "optimal" time) or it will select new values for num_gangs and vector_length, repeating the process from Step 1.

In Step 3, the tuner executes the target program and extracts timing information. It is important to note that *the tuner does not measure the running time of the target program; rather, it is the target program's responsibility to measure the execution time of the affected kernel(s) and report it to the autotuner.* This can be done in one of two ways.

- **The program can be compiled with the PGI compiler's `-ta=nvidia,time` flag**, which links the program with a profile library that collects and displays timing information about accelerator regions and writes it to standard error after the program runs.
- **The user can insert timing code into the program and print the results to standard output.** A typical approach would be to insert calls to `omp_get_wtime()` above and below a parallel loop, compute the difference `d` of the two times, and then invoke `printf("time=%f\n", d)`. (If the timing output does not have this exact form, the autotuner can be given a command line flag with a regular expression providing an alternative pattern to match.)

The tuner will run the program multiple times and collect the average and standard deviation of the reported times. (Again, the number of repetitions is configurable.) After autotuning completes, the tuner displays the results, including the timing information for each (num_gangs, vector_length) pair considered and the best result discovered (see Figure 1). Results can also be written to a CSV file, Excel spreadsheet, or gnuplot script for further analysis.

4.1 Search Methods

The tuner supports multiple search methods, including exhaustive searches (grid searches) and direct search methods. Table 1 shows the search methods currently supported by the autotuner, along with the number of (num_gangs, vector_length) pairs tested before termination and what restrictions are placed on the candidate values for num_gangs and vector_length.

Exhaustively testing a range of parameter values is an effective way to determine good values for num_gangs and vector_length. This is sometimes called a *grid search* or *parameter sweep* and is implemented by the *grid* methods in Table 1. However, it requires compiling and running the target program many times. For example, testing all multiples of 32 between 32 and 1024 requires compiling and running

Search Method	Num. Points Tested	Values Considered	
		num_gangs	vector_length
grid32	1024	Multiples of 32	Multiples of 32
grid64	256	Multiples of 64	Multiples of 64
grid128	64	Multiples of 128	Multiples of 128
grid32-vlpow2	320	Multiples of 32	Powers of 2
grid-pow2	100	Powers of 2	Powers of 2
nelder-mead	Varies (avg. 7)	Candidate multiples of 32	Candidate powers of 2
coord-search	Varies (avg. 11)	Candidate multiples of 32	Candidate powers of 2

Table 1: Search methods supported by the OptACC autotuner.

the program 1024 times—impractical for most applications.

Therefore, it is preferable to use a more advanced search technique that can evaluate just a few points and use timing data from those points to efficiently determine which other points should be tested and which can be ignored. This is the goal of the Nelder-Mead and Coordinate Search methods.

4.2 Autotuning as an Optimization Problem

To approach the problem of autotuning OpenACC kernels, we model it as a mathematical optimization problem

$$\begin{array}{ll} \text{minimize} & f(g,v) \\ \text{subject to} & g_{\min} \le g \le g_{\max} \\ & v_{\min} \le v \le v_{\max}, \end{array}$$

where g represents the value of num_gangs, v represents the value of vector_length, and $f(g,v)$ is the objective function corresponding to the OpenACC kernel being tuned. In our case, the objective is to minimize the runtime of the kernel, so $f(g,v)$ maps (num_gangs, vector_length) pairs to the runtime of the kernel when compiled using those parameters. $g_{\min}$ and $g_{\max}$ are respectively the minimum and maximum allowable values for num_gangs, and may be configured by the user or left as default values 2 and 1024. $v_{\min}$ and $v_{\max}$ are respectively the minimum and maximum allowable values for vector_length and are again configurable, with defaults 2 and 1024.

We can visualize the behavior of the objective function $f(g,v)$ by treating (num_gangs, vector_length) pairs as points in the Cartesian plane and associating with each point the runtime of the kernel, giving a three-dimensional surface plot. Figure 2 gives the surfaces plotted from the exhaustive test data for several benchmarks.

Optimization is a well-studied problem in mathematics, however our objective function $f(g,v)$ presents several challenges for standard optimization methods:

- Evaluating $f(g,v)$ is time-consuming, since it requires recompiling and running the program.
- n and v cannot take arbitrary real values. Setting num_gangs or vector_length to 32.6 or -10, for example, is not possible. In fact, we often add additional constraints; for example, we may restrict v to multiples of 32 or powers of 2 (see Table 1).
- Derivative information about $f(g,v)$ is unavailable.

We focus therefore on a special class of optimization methods called *direct search* methods, which seek optimal solutions while relying only on evaluations of the objective function. For our tuner, we implemented two well-known direct search methods: a modified Nelder-Mead method and a Coordinate Search method. To optimize an objective over n variables, the Nelder-Mead algorithm uses a simplex of $n+1$ vertices; on each iteration, the simplex is transformed by reflection, expansion, contraction, and shrinkage based on the values of selected points in or calculated from the simplex [13, 17]. The method terminates when the points in the simplex are arbitrarily close together and the evaluations of the objective function at the points in the simplex are arbitrarily close.

Our objective function is not continuous over the real numbers, since num_gangs and vector_length are required to be integers, and in some cases they are restricted to powers of 2 (due to a PGI compiler bug). Therefore, we implemented a modified Nelder-Mead method that considers only those points which correspond to valid (num_gangs, vector_length) pairs, using an adaptation of the modifications presented in [6]. Any time the modified Nelder-Mead method calculates a new point to test, the point is rounded to the nearest valid (num_gangs, vector_length) pair. Since we restrict the points in the simplex by rounding, the termination condition described above is not applicable to our problem. Instead, we consider the method to have converged if the same point appears more than once in the simplex.

The new points calculated in each step of the Nelder-Mead algorithm are controlled by the values of the constants for reflection, expansion, contraction, and shrinkage ρ, χ, γ, and σ, respectively. For our implementation, we chose

$$\rho = 1, \chi = 2, \gamma = 0.5, \text{ and } \sigma = 0.5.$$

As noted in [17], this choice of values is nearly universal.

The Coordinate Search method (in two dimensions) starts with an initial point x and an initial step size $\alpha > 0$. (Our implementation uses $\alpha = 256$.) It evaluates f at the four points α units above, below, and to the left and right of x. If any of the four points results in a smaller value of f, then x is reset to that point, and the process repeats. If none of the four points produces a smaller value of f, the step size α is reduced, and x remains unchanged. Our implementation reduces the step size to 0.75α and terminates after two consecutive unsuccessful iterations. Both the Nelder-Mead and Coordinate Search implementations use 256 gangs and a vector length of 128 as their initial point (the PGI compiler frequently chooses these as defaults).

Conceptually, both Nelder-Mead and Coordinate Search sample values of the objective function near a particular point, move in a direction that appears likely to lead "downhill" from that point, and repeat. Such methods are heuristic and are certainly not guaranteed to find a global minimum. Nevertheless, visualizing walking downhill on the surfaces in Figure 2 suggests why these search methods are likely to be successful: With the right starting point, walking downhill will almost always lead to a good, if not optimal, value. In particular, many of the 36 kernels we tested had a shape comparable to that of the *covariance 2* benchmark, which is very amenable to this type of search.

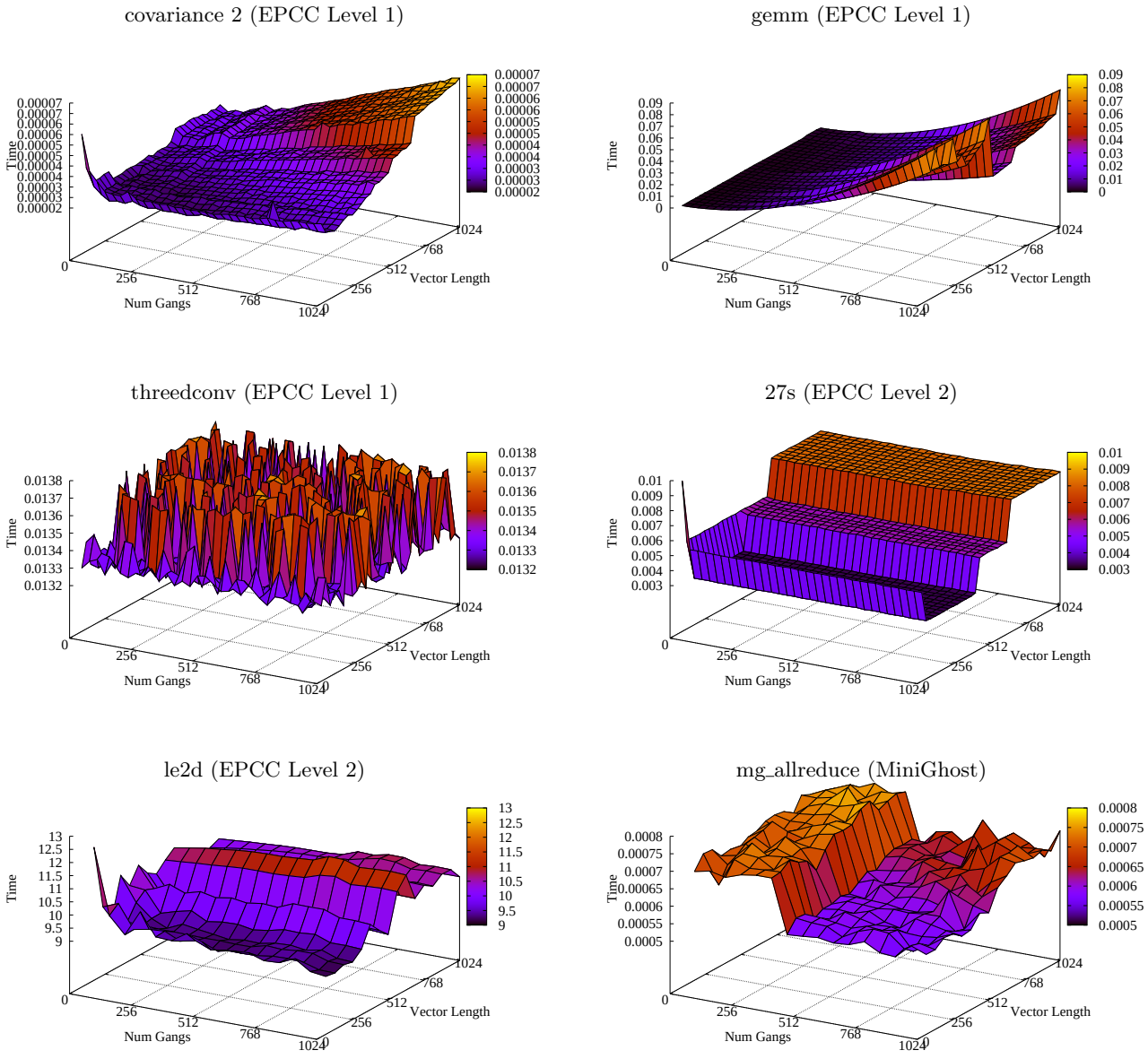


Figure 2: Examples of surfaces formed from (num_gangs, vector_length, time) triples.

5. EVALUATION & DISCUSSION

We aimed to answer the following questions:

- **RQ1: Benefit.** Does tuning the number of gangs and vector length produce a speedup?
- **RQ2: Quality.** Can the autotuner find high-quality values for num_gangs and vector_length with only a few runs of the target program? How do the number of gangs and vector length found by direct search methods compare to optimal values found by a grid search?

5.1 Test Codes & Results

To answer these questions, we evaluated our autotuner on 36 kernels: 25 from the EPCC OpenACC Benchmarks [1,12] and 11 from an OpenACC translation of MiniGhost [7]. All kernels were compiled using the PGI Accelerator[3] and run on DMC[4]. DMC's Kepler nodes have 2.5 GHz Intel Xeon (Ivy Bridge) CPUs, 128 GB of RAM, and Tesla K20m GPUs.

To assess the quality of the Nelder-Mead and Coordinate Search results, we obtained bounded exhaustive timing data for each kernel to use for comparison. We used OptACC's *grid32* grid search strategy when possible. For time-consuming kernels, we used the *grid64* strategy. A bug in the PGI compiler caused a runtime failure when a kernel

[3]pgcc 14.9-0 64-bit target on x86-64 Linux -tp nehalem.
[4]https://www.asc.edu/html/dmc.shtml. At the time of data collection, Comet (the only XSEDE resource supporting OpenACC) was not yet online. DMC's Kepler GPUs are comparable to those on Stampede.

with a `reduction` clause had a vector length that was not a power of two; therefore, some EPCC Level 1 benchmarks had to be evaluated using the *grid32-vlpow2* strategy.

To obtain timing data for each kernel, we inserted calls to `omp_get_wtime` above and below each parallel loop, inside the data directive if one was given. (The timer has a microsecond resolution according to `omp_get_wtick`.) Using OpenMP timers (as opposed to the timing data produced by the PGI compiler's `-ta=nvidia,time` option) allowed us to measure the execution time of the specific parallel loop being tuned.

The results of our evaluation are shown in Figure 3. The bar graphs at the top show the average running times for each kernel (with standard deviations indicated by error bars). The *Untuned* bar shows the kernel's execution time with the `num_gangs` and `vector_length` clauses omitted, in which case the compiler selects default values. The *Nelder-Mead* and *Coord. Search* bars show the runtimes after autotuning using the respective direct search method. The *Grid Search* bar shows the smallest execution time obtained by a grid search (i.e., a bounded exhaustive search of feasible parameter values), which provided a basis of comparison for the more efficient search methods.

The columns of the table are as follows:

- **Kernel** provides the name of the benchmark and, if there were multiple kernels in a file, the (sequential) number of the kernel that was tuned.
- **Grid Search** identifies the method used to obtain bounded exhaustive timing data.
- The **Speedup** columns show the speedup from the untuned time to the best time found by each method.
- After the direct search methods found a local optimum, we compared the timing data for that point to the exhaustive timing data. The **Pct** column shows the percentile of the point obtained by the direct search method, with respect to the exhaustive timing data.
- The **Count** column reports the number of points tested by each direct search method, i.e., the number of times the target program had to be compiled and executed.
- **Better** indicates which direct search method found the higher-quality point (by percentile). If both methods found points in the same percentile, the method that tested fewer points is shown.

5.2 RQ1: Benefit

Does tuning the number of gangs and vector length produce a speedup?

In Figure 3, the **Grid Speedup** column shows the speedup of the best point found by a grid search. This is, in theory, the maximum speedup that could possibly be obtained by either Nelder-Mead or Coordinate Search. The speedups of those methods are shown under their respective headings.

The EPCC Benchmarks are divided into two groups. The Level 1 benchmarks are BLAS-like kernels, ported from the Polybench and Polybench/GPU benchmark suites. Eleven of the 22 Level 1 benchmarks were able to achieve a speedup greater than 2×. Three kernels achieved speedups above 5×; in all three cases, the optimal number of gangs was quite large (the PGI compiler tends to choose smaller numbers as defaults). The three Level 2 benchmarks are application codes. Although the runtime of the *le2d* benchmark was improved by a few milliseconds, varying the number of gangs and vector length had very little impact on the performance of the Level 2 benchmarks overall.

MiniGhost contained OpenACC kernels in three files. The kernel from `MG_ALLREDUCE` (a sum reduction) achieved a significant speedup, as did two of the six kernels from `MG_PACK` (which are essentially data copies). MiniGhost's four stencil computation kernels occupy the majority of the GPU time, according to a profile, but the compiler's default choices were good; none of them benefited from autotuning.

5.3 RQ2: Optimality

Can the autotuner find high-quality values for num_gangs and vector_length with only a few runs of the target program? How do the number of gangs and vector length found by direct search methods compare to optimal values found by a bounded exhaustive (grid) search?

The goal of the Nelder-Mead and Coordinate Search methods is to find high-quality points while testing relatively few points. The last four columns in Table 1 show the quality of the results obtained by these methods.

With the Nelder-Mead search:

- In all but four cases, the suggested point was in the top 25% of points found by a grid search.
- In 19 of the 36 kernels, the point was in the top 5%.
- The search converged after evaluating 7 points, on average, and 24 points in the worst case.

With the Coordinate Search:

- All points found were in the top 25% of points found by a grid search.
- In 22 of the 36 kernels, the point was in the top 5%.
- The search converged after evaluating 11 points, on average, and 20 points in the worst case.

Thus, both methods were successful in finding high-quality points. However, as the last column in Table 1 indicates, neither method was universally better than the other.

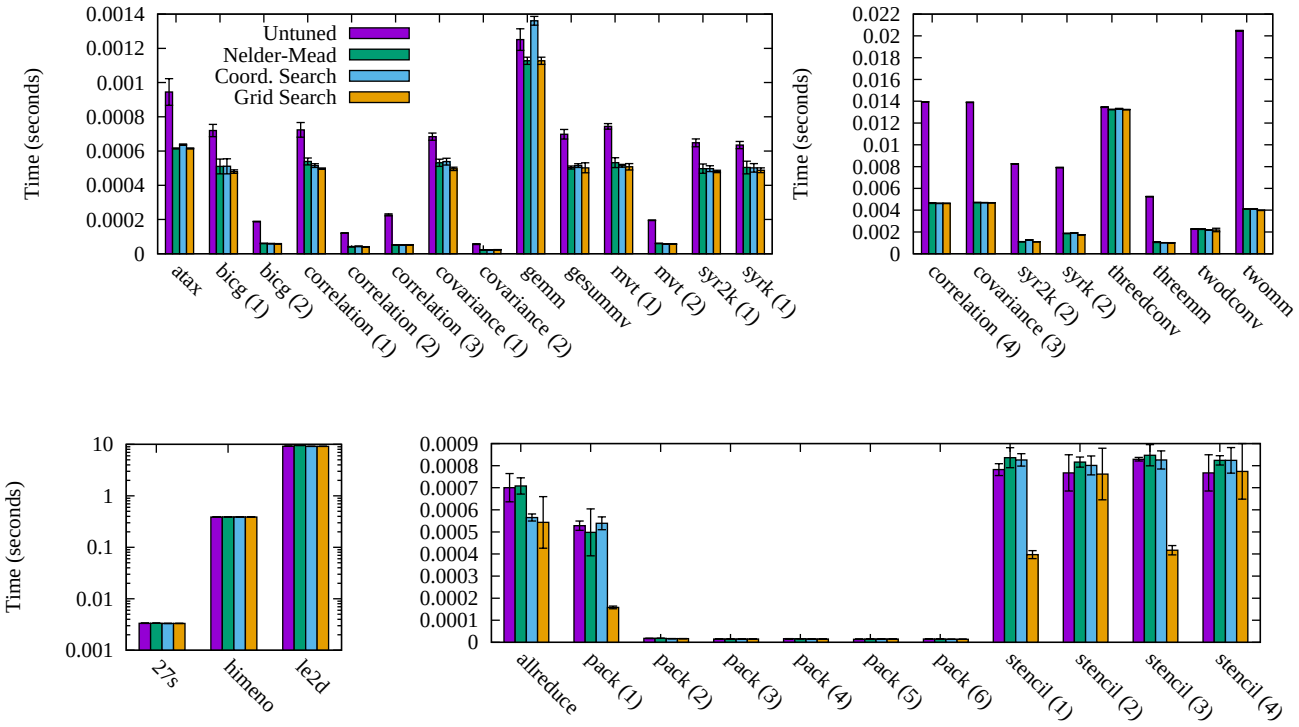
5.4 Limitations

Neither Nelder-Mead nor Coordinate Search is guaranteed to find a global minimum, and the exact local minimum found depends heavily on the choice of an initial simplex/initial point and other parameters. Our choices worked well on the 36 kernels we tested. Although the intent of a benchmark suite, such as the EPCC OpenACC benchmarks, is to be representative of a larger class of codes, this is not guaranteed. For example, some MiniGhost kernels performed well with very small numbers of gangs—smaller than any of those required by the EPCC suite. The autotuner's algorithms may benefit from additional modifications as it is employed on a larger number of applications.

In addition, the authors had access to only one production OpenACC compiler—PGI Accelerator[5]—and focused on one NVIDIA GPU. The hardware and compiler both impact the shape of the autotuning surface.

Finally, while some of the kernels obtained impressive speedups, the reader is cautioned that these are only *kernel* execution times, which in most cases were just a few milliseconds (e.g., *syr2k_k2* dropped from 8.26 ms to 1.10 ms). In a larger application, accelerator kernels are often just a fraction of the total runtime. Amdahl's Law always applies.

[5]Unfortunately, CAPS is no longer in business.



	Kernel	Grid Search	Grid Speedup	Nelder-Mead Speedup	Nelder-Mead Pct	Nelder-Mead Count	Coord. Srch. Speedup	Coord. Srch. Pct	Coord. Srch. Count	Better
EPCC Benchmarks – Level 1	atax	32-vlpow2	1.54×	1.54×	1%	10	1.48×	10%	13	NM
	bicg (1)	32-vlpow2	1.50×	1.41×	5%	13	1.41×	6%	13	NM
	bicg (2)	32-vlpow2	3.27×	3.16×	14%	4	3.17×	9%	11	CS
	correlation (1)	32-vlpow2	1.46×	1.34×	30%	4	1.40×	8%	16	CS
	correlation (2)	32-vlpow2	2.97×	2.90×	2%	4	2.76×	13%	8	NM
	correlation (3)	32	4.39×	4.42×	4%	6	4.45×	1%	20	CS
	correlation (4)	32-vlpow2	3.02×	3.00×	7%	4	3.02×	0%	20	CS
	covariance (1)	32-vlpow2	1.38×	1.29×	7%	10	1.27×	14%	10	NM
	covariance (2)	32	2.49×	2.56×	0%	6	2.56×	0%	12	NM
	covariance (3)	32-vlpow2	2.97×	2.95×	8%	4	2.96×	4%	8	CS
	gemm	32	1.11×	1.11×	0%	24	0.92×	1%	15	NM
	gesummv	32-vlpow2	1.39×	1.38×	1%	4	1.36×	3%	11	NM
	mvt (1)	32-vlpow2	1.47×	1.40×	7%	9	1.45×	1%	8	CS
	mvt (2)	32-vlpow2	3.48×	3.25×	5%	10	3.36×	18%	4	NM
	syr2k (1)	32	1.35×	1.30×	2%	7	1.30×	2%	9	NM
	syr2k (2)	32	7.49×	7.47×	3%	7	6.51×	10%	8	NM
	syrk (1)	32	1.30×	1.26×	8%	8	1.26×	4%	11	CS
	syrk (2)	32-vlpow2	4.54×	4.25×	2%	8	4.15×	10%	11	NM
	threedconv	32	1.02×	1.02×	0%	20	1.01×	22%	8	NM
	threemm	32-vlpow2	5.24×	4.92×	13%	6	5.15×	1%	8	CS
	twodconv	32	1.04×	1.01×	8%	4	1.04×	0%	9	CS
	twomm	32-vlpow2	5.13×	5.00×	6%	4	4.98×	9%	11	NM
Lvl 2	27s	32	1.01×	1.00×	20%	4	1.01×	1%	16	CS
	himeno	32	1.00×	1.00×	1%	6	1.00×	2%	16	NM
	le2d	64	1.01×	0.97×	24%	5	1.01×	1%	8	CS
MiniGhost	allreduce	64	1.29×	0.99×	68%	6	1.24×	4%	16	CS
	pack (1)	64	3.35×	1.06×	2%	6	0.98×	5%	8	NM
	pack (2)	64	1.05×	0.99×	5%	6	1.04×	1%	11	CS
	pack (3)	64	1.05×	1.05×	0%	6	1.01×	2%	11	NM
	pack (4)	64	1.12×	1.03×	3%	6	1.09×	1%	11	CS
	pack (5)	64	0.96×	0.96×	2%	6	0.96×	2%	11	NM
	pack (6)	64	1.10×	1.01×	5%	6	1.10×	0%	11	CS
	stencil (1)	64	1.97×	0.94×	38%	6	0.95×	23%	8	CS
	stencil (2)	64	1.01×	0.94×	10%	5	0.96×	2%	11	CS
	stencil (3)	64	1.99×	0.98×	30%	5	1.00×	12%	12	CS
	stencil (4)	64	0.99×	0.93×	7%	6	0.93×	8%	11	NM

Figure 3: Evaluation results. The complete data set is available at `https://github.com/OptACC/Benchmark-Data`.

6. FUTURE WORK

Our autotuner is open source software and is available on GitHub at https://github.com/OptACC/OptACC. We have tested it on Linux systems using Python 2.6.9, 2.7.3, and 3.4.3. The complete evaluation data set, including surface plots for all kernels (as in Figure 2), is available from https://github.com/OptACC/Benchmark-Data.

There are a number of directions for future work.

Support source code modification. OptACC currently requires that the user manually prepare files for tuning by adding num_gangs and vector_length clauses to their OpenACC `parallel` directives. The user must also add timing code manually if PGI's timers are not used. In the future, the tuner could insert appropriate clauses and timing code directly into the user's source code. This would be especially valuable in an IDE like Eclipse, where the user could select a loop and autotune it with a single mouse click.

Autotune for worker parallelism. We have omitted tuning num_workers, since is infrequently used with the PGI compiler and NVIDIA GPUs, although the autotuner would benefit from its addition, particularly when used with other compilers and accelerators.

Integrate static analyses. Autotuning could also benefit from the integration of a parser and static analyses. For example, a loop nest with only gang and vector loops does not need to be tuned for worker parallelism, and *a priori* knowledge of the number of loop iterations could be used to inform the search (e.g., by changing the initial simplex for the Nelder-Mead algorithm).

Integrate compiler diagnostics. Currently, the autotuner is unaware of what num_gangs and vector_length values were chosen by the compiler as its defaults. In the future, it may be beneficial to use the compiler's values to seed the autotuning process.

Autotune across multiple input sizes. As other authors have noted [8], autotuning results for one input size are not necessarily optimal for other input sizes. Autotuning across multiple input sizes without the benefit of historic or training data remains an open problem.

Integrate other transformations. The possibilities for autotuning OpenACC code extend far beyond simply tuning the number of gangs and the vector length of a parallel region. As other authors have noted, an autotuner could also place *gang*, *worker*, and *vector* clauses in a loop nest [15] or even make other compiler-like transformations to the program [6]. These are significantly more difficult search problems, but they are also problems for which a breakthrough result would be extremely valuable.

Acknowledgments

This work used the Extreme Science and Engineering Discovery Environment (XSEDE) [16], which is supported by National Science Foundation grant number ACI-1053575. This work was also made possible by a grant of high performance computing resources and technical support from the Alabama Supercomputer Authority.

7. REFERENCES

[1] EPCC OpenACC benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite.

[2] OpenACC Directives | Developing with GPUs | NVIDIA. http://www.nvidia.com/object/openacc-gpu-directives.html.

[3] OpenACC: Directives for accelerators. http://www.openacc-standard.org/.

[4] The OpenACC application programming interface, version 2.0a. http://www.openacc.org/sites/default/files/OpenACC.2.0a_1.pdf, August 2013.

[5] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarisinghe. OpenTuner: An extensible framework for program autotuning. *PACT '14*, 2014.

[6] P. Balaprakash, S. M. Wild, and P. D. Hovland. Can search algorithms save large-scale automatic performance tuning? *Procedia Computer Science*, 4:2136–2145, 2011. Proc. ICCS 2011.

[7] R. F. Barrett, C. T. Vaughan, and M. A. Heroux. MiniGhost: A miniapp for exploring boundary exchange strategies using stencil computations in scientific parallel computing. Technical Report SAND2012-2437, Sandia National Laboratories, 2012.

[8] L. Cebamanos. Autotuning NekBone for OpenACC, July 2014. https://www.epcc.ed.ac.uk/blog/2014/07/03/autotuning-nekbone-openacc.

[9] M. Colgrove. PGInsider March 2012: 5x in 5 hours: Porting a 3D elastic wave simulator to GPUs using PGI Accelerator, March 2012. http://www.pgroup.com/lit/articles/insider/v4n1a3.htm.

[10] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. SIAM, Philadelphia, PA, USA, 2009.

[11] M. Frigo and S. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, Feb 2005.

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

[13] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer J.*, 7:308–313, 1965.

[14] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. SPIRAL: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, Feb. 2004.

[15] S. Siddiqui and S. Feki. Historic learning approach for auto-tuning OpenACC accelerated scientific applications. In *International Workshop on Automatic Performance Tuning*, Eugene, Oregon, July 2014. King Abdulla University of Science and Technology.

[16] J. Towns, T. Cockerill, M. Dahan, I. Foster, K. Gaither, A. Grimshaw, V. Hazlewood, S. Lathrop, D. Lifka, G. D. Peterson, R. Roskies, J. R. Scott, and N. Wilkens-Diehr. XSEDE: Accelerating scientific discovery. *Computing in Science and Engineering*, 16(5):62–74, 2014.

[17] M. H. Wright. Direct search methods: Once scorned, now respectable. In D. Griffiths and G. Watson, editors, *Numerical Analysis 1995 (Proc. 1995 Dundee Biennial Conf. in Numerical Analysis)*, pages 191–208, Harlow, UK, 1995. Addison Wesley Longman.