Title

Toward Techniques for Auto Tuning GPU Algorithms

Authors

Davidson, Andrew

Owens, John D

Publication Date

2026-09-11

Toward Techniques for Auto-Tuning GPU Algorithms

Andrew Davidson * and John D. Owens †

University of California, Davis

Abstract We introduce a variety of techniques toward auto-tuning data-parallel algorithms on the GPU. Our techniques tune these algorithms independent of hardware architecture, and attempt to select near-optimum parameters. We work towards a general framework for creating auto-tuned data-parallel algorithms, using these techniques for common algorithms with certain characteristics. Our contributions include tuning a set of algorithms with a variety of computational patterns, with the goal in mind of building a general framework from these results. Our tuning strategy focuses first on identifying the computational patterns an algorithm shows, and then reducing our tuning model based on these observed patterns.

1 Introduction

Given the complexity of emerging heterogeneous systems where small parameter changes lead to large variations in performance, hand-tuning algorithms have become impractical, and auto-tuning algorithms have gained popularity in recent years. When tuned, these algorithms select near-optimum parameters for any machine. We demonstrate a number of techniques which successfully auto-tune parameters for a variety of GPU algorithms. These algorithms are commonly used, were not synthetically created and were chosen to rely on different parameters and be performance bound in different ways.

Both work by Liu et al. [4] and Kerr et al. [2] use adaptive database methods for deriving relationships between input sets and their direct influence on the parameter space. Work by Li et al. [3] focuses on tuning one GEMM algorithm, while work by Ryoo et al. [5] mainly deals with kernel and compiler level optimizations for one machine (an 8800GTX). Our work differs from these in that we first try to identify computational patterns algorithms have and then tune their parameters using a variety of methods. Rather than use a single data-base method which might be susceptible to nearly impossible to predict non-linear input to parameter behavior, we try to identify what behavior the algorithm has to the input set, and choose a tuning strategy accordingly. This allows us to heavily prune the parameter search space, before attempting to tune the algorithm, resulting in very fast tuning runs. For our set of algorithms, we showed a quick tuning run (less than a few minutes for most machines) can generate the needed information to automatically choose near-optimum parameters for future runs.

2 Algorithms

When attempting to auto-tune an algorithm we approach the problem using this philosophy:

- Identify the tunable parameters for the algorithm.
- Look for a relationship between the input space, and the parameter space. As an example, a relationship between these two sets was seen in our reduction and scalar product kernels. As memory bound processes, the larger the work load (input set) the more threads (parameter set) were required until a thread cap was reached. Sections 2.1 and 2.2 covers this strategy in more depth.
- If such a relationship exists, we must build a model from your input space to parameter space, or model the parameter space solely on machine characteristics.

We studied four parallel algorithms extensively in our work. Reduction, a common parallel primitive which operates on a large set of values and reduces that set to one value. Next, scalar product, which sums up the products between two vectors. Third, an N-Body algorithm which simulates the gravitational interactions between a set of objects. Finally, SGEMM, a fast matrix multiply. For the reduction, scalar product and N-Body algorithms we used the kernels found in the NVIDIA SDK as our base algorithms. These algorithms have already been highly optimized, and are considered standards for benchmarking. Since many of these algorithms have been hand-tuned for certain machines, we may receive only a marginal performance boost[1]. For our SGEMM algorithm we used Lung-Shen Chien's highly tuned and optimized kernel [1] which demonstrates higher performance than that released by NVIDIA's SGEMM. In our final auto-tuned algorithm we wish to trim as much of the parameter search space as possible, while maintaining an accurate model to select parameters. Auto-tuning algorithms that use standard techniques such as model driven inputs, with interpolation

*Email: aaldavidson@ucdavis.edu
†Email: jowens@ece.ucdavis.edu

[1] In particular, the NVIDIA SDK has had a number of changes to obtain optimal performance on GT200 series cards

between points, may result in sub-optimal parameters (one method does not fit all). This is where the major challenge lies, as great care must be taken to ensure your auto-tuning strategy is not susceptible to unpredictable non-linear effects.

2.1 Reduction

We made a few minor improvements to NVIDIA's SDK reduction kernel code. We chose this algorithm as it is obviously memory bound and each item can be accessed in any order (coalesced reads would obviously be preferred). Therefore tuning reduction would give insight into many other common parallel algorithms, such as scan and sort, which have similar computational patterns. For the reduction kernel, the parameters that require tuning are the number of threads per block, and the number of blocks. The optimum set of parameters may fluctuate with respect to the number of elements we reduce.

Through experiments and benchmarking, we were able to show a standard behavior for the optimum parameter set given a number of elements. Using the results from these tests, our auto-tuning method first searches for a thread cap for all elements, which is assumed to fully occupy the machine. The thread cap is defined as the maximum number of threads optimum to any valid input set (no optimum parameters will have more total threads than the thread cap).

Since our input set is a one-dimensional array in the reduction algorithm, it is easy to test what this thread cap is, and where it applies. All workloads greater than the number of elements where this thread cap applies, is also bound by the thread cap. Next we test for a lower switch-point where the CPU outperforms the GPU, and the optimum parameters for that point. Using these two points, and their associated number of elements, we are able to select a number of total threads (threads per block and number of blocks) for any input set.

2.2 Scalar Product

This algorithm is also memory bound and available in NVIDIA's SDK. However the parameters for tuning the Scalar Product kernel are more complex as the vector size and total number of vectors adds a dimension to the tuning process. Therefore while applying some of the same principles from our reduction method for a thread cap and associated work cap, we also select a set of random points ($vsize_i$, n_i) such that $vsize_i < vsize_m$ and $n_i < n_m$, that are under the work cap and test for the optimum number of threads. For all points under the thread cap we select the parameters for the knot *closest* to that data-point. We used this approach rather than an interpolation approach, as our tests showed the closest knot approach performed better and more predictably.

2.3 N-Body

We selected the N-Body algorithm as it has a number of tunable parameters, is more arithmetically intense, and

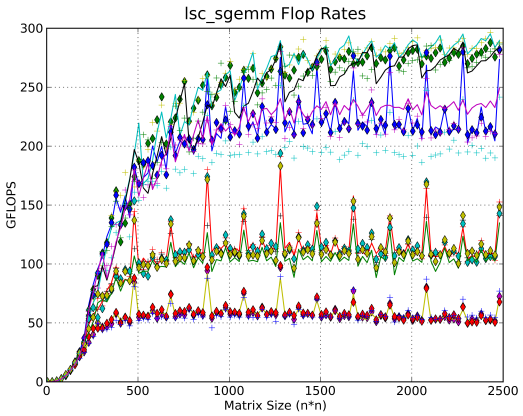

Figure 1: Shows a brute force test of the twenty-one configurations we created of Chien's SGEMM code [1]. Each line represents the flop rate of one configuration, with the default line being teal. All of these were run on a GTX 260. A select few kernels perform well, while others suffer poor results, however once a suitable kernel is chosen for a machine, it can be used for all input sets with near-optimum results.

therefore not global memory bound. The tunable parameters in this case are two dimensions of block sizes that will be shared per block. Therefore tuning these parameters involves a tradeoff in register usage, and the amount of data sharing per block. This optimum point does not vary widely when changing the number of bodies. Using this information, our auto-tuning does not take variability in the input set into account. Rather we can tune for one non-trivial input set and reuse these parameters for future runs, greatly reducing the tuning search space.

2.4 SGEMM

As mentioned previously we used Lung Sheng Chien's optimized code [1] (which is a further optimization of Vasily Volkov's code [6]) as the base kernel for our auto-tuning method. There were a number of variations to this set of code. We selected the one with best performance, that would not lead to out-of-bound problems (method8 in Chien's benchmark suite). Using templates, we were able to form twenty-one valid versions of this method that relied on three input parameters. Due to Chien's reliance on pre-compiling kernels into cubin binaries before runs, we created twenty-one associated binaries for each version. Experimental results showed that a number of these valid versions were inherently inefficient, and were removed from the tuning suite.

Like the N-Body algorithm, if an optimum parameter set is found for some matrix size (number of bodies in the N-Body case), it generally applies for all input sets on that machine. Therefore, our strategy was to find the optimum parameter set (of the twenty-one available) for a given machine which would allow us to tune all input-sets. It was also noticed that in general, the kernels which had higher register usage per thread, performed better. Whether or not

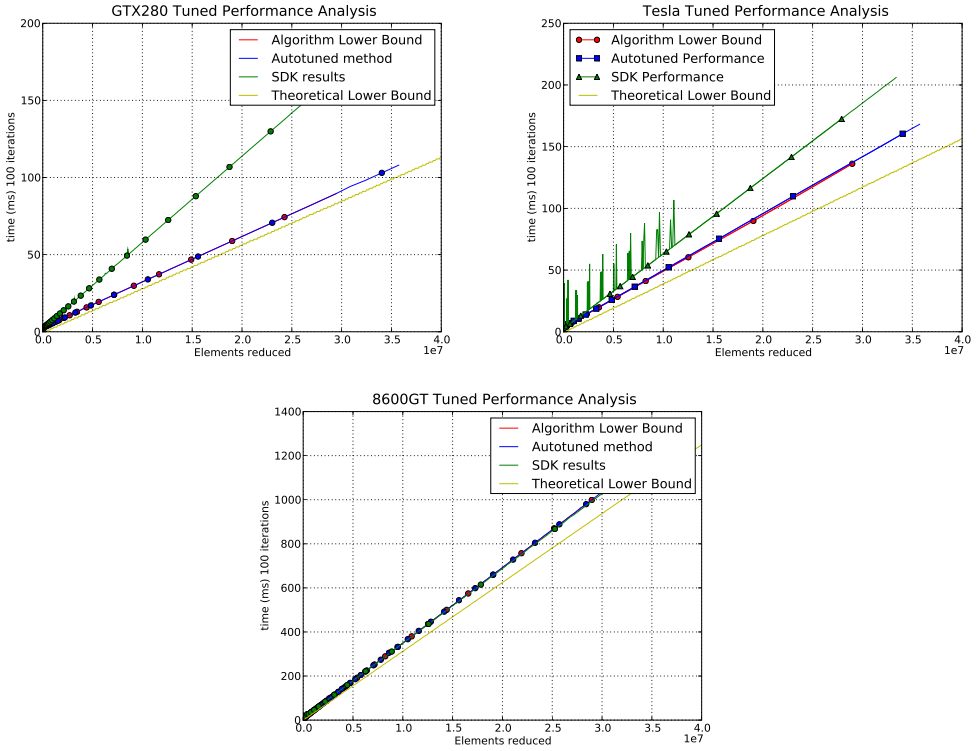

Figure 2: Demonstrates the bandwidth performance of our auto-tuned reduction method versus that of the SDK, with included theoretical and actual algorithm bounded limits. These tests were run on three cards: a Tesla C1060, a GTX 260 and a 8600 GTS.

this is a rule of thumb one can follow for quickly selecting an appropriate variation, requires more testing on a wider variety of devices.

3 Results

We tested our methods, and ran a number of experiments on a higher end GTX 260, and a low-end 8600 GTS. We had access to a Tesla C1060 for our reduction tests, however it was unavailable for test runs for our other algorithms[2]. However, since these tuning methods have worked correctly for these cards with such a large difference in capability (30 SMs vs 4 SMs), we believe they should also work adequately for mid-range cards.

3.1 Reduction

Figure 2 compares the performance of our auto-tuning method against that of the SDK's default parameters. The results show a speedup that brings memory performance close to both the bandwidth limit. Our auto-tuned method(blue plot) performs as well as a brute force check on all possible parameters (red line), while only taking a minute or less to run a one-time tuning run. As a memory bound function, we found performance depended directly on the number of memory controllers available to the machine. Though this can not be queried directly, some pre-tuning tests can supply this information, and be used to reduce the tuning space further[3].

3.2 Scalar Product

Though using knots adds complexity and possibility for non-optimal selections, our results still perform better than that of NVIDIA's SDK for most points. Since visualizing the performance of a multi-dimensional input space is difficult, we instead present our results in Table 1 as the performance from a set of random inputs. Our results show a speedup for almost all cases on the GTX 260 (achieving a 15% performance boost). While the performance gains on the 8600GTS were not as drastic we still were able to boost performance slightly.

3.3 N-Body

We ran our auto-tuning algorithm which searches one input set for the optimal parameter set (p,q) which is the number of shared elements communicating in a block. Further details on these parameters can be found in the NVIDIA CUDA SDK documentation. Our auto-tuned method had a 3.4% flop rate increase over the SDK's default parame-

[2]We are currently in the process of receiving mid-range cards for testing purposes, and will have results for them shortly.

[3]Since these tests and graphs were produced, NVIDIA has updated its default parameters to run well on their GT200 line. However, this means that older cards, such as the GT80 series, run poorly with these parameters. We believe this highlights the importance of auto-tuning, as machine-dependent parameters become more apparent.

Scalar-Product Auto-Tuning Comparison			
Device	Overall Speedup	Worst Case	Best Case
GTX260	1.15	.976	1.1635
8800GTS	1.0092	.9918	1.043

Table 1: We performed this test on a set of 1000 random input sets. The *Overall Speedup* is the ratio of the sum of all times, while *Worst Case* and *Best Case* ratios are the worst individual and best individual ratios of the random input set.

N-Body Auto-Tuning Comparison			
Device	Untuned	Tuned	Perf. Increase
GTX260	320.758 GF/s	331.753 GF/s	1.034
8800GTS	37.518 GF/s	39.992 GF/s	1.066

Table 2: Shows the performance increase for an auto-tuned N-Body simulation, versus an untuned run. The number of objects in this run was $24,134$, however the performance increase remains fairly constant between input sets.

ters on the GTX260, and a 6.6% increase on the 8600GTS. Table 2 contains more detailed performance data for our method and that of NVIDIA's default parameters.

3.4 SGEMM

As was illustrated in Section 2.4, small variations in parameters could lead to large variations in kernel performance. However, the default parameters that Chien [1] used were found to be the optimal kernels for both the GTX260 and 8600GTS. Whether this holds true for other cards requires tests on a broader range of devices.

4 Conclusion

We believe that hand-tuning algorithms for each machine will become an impractical method as systems become more diverse in capability, and algorithm bounds become more complex. Therefore, developing methods that automate the tuning process could prove to be a powerful tool to boost utilization. Our work has shown a number of auto-tuning methods which boost performance for a number of common algorithms. Though continued work is needed, we believe this is an important stepping stone to providing a generalized tuning methodology for data parallel programs.

References

[1] L.-S. Chien. Hand-tuned SGEMM on GT200 GPU. Technical report, Tsing Hua University, 2010. `http://oz.nthu.edu.tw/~d947207/NVIDIA/SGEMM/HandTunedSgemm_2010_v1.1.pdf`.

[2] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. In *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 31–42, New York, NY, USA, 2010. ACM.

[3] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning gemm for gpus. In *ICCS '09: Proceedings of the 9th International Conference on Computational Science*, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu program optimizations. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[5] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *CGO '08: Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 195–204, Apr. 2008.

[6] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.