

Automatic Stencil Code Generation-

Ph.D. Thesis Proposal

Kaushik Datta
kdatta@cs.berkeley.edu

March 2, 2007

Abstract

Stencil-based kernels constitute the core of many scientific applications on block-structured grids. These calculations form the basis for a wide range of scientific applications from simple Jacobi iterations to complex multigrid and block structured adaptive PDE solvers. Unfortunately, these codes achieve a low fraction of peak performance, due primarily to the disparity between processor and main memory speeds. I propose for my Ph.D. dissertation research to develop an automatic system to generate highly efficient, platform-adapted implementations of stencil kernels. In practice, performance is a complex function of many factors, including compiler technology, machine architecture, instruction scheduling, and memory access behavior. However, through the use of performance models and search, we can generate very good, if not optimal stencil code. This tuned code can be over twice as fast as untuned code.

1 Introduction

Partial differential equation (PDE) solvers constitute a large fraction of scientific applications in such diverse areas as heat diffusion, electromagnetics, and fluid dynamics. These applications are often implemented using iterative finite-difference techniques, which sweep over a spatial grid, performing nearest neighbor computations called *stencils*. In a stencil operation, each point in a multidimensional grid is updated with weighted contributions from a subset of its neighbors in both time and space—thereby representing the coefficients of the PDE for that data element. These operations are then used to build solvers that range from simple Jacobi iterations to complex multigrid and adaptive mesh refinement methods [2].

Stencil computations perform global sweeps through data structures that are typically much larger than the capacity of the available data caches. As a result, stencil computations generally achieve a low fraction of theoretical peak performance, since data from main memory cannot be transferred fast enough to avoid stalling the functional units on modern microprocessors. To remedy this, optimizations include dividing the grid into blocks or tiles, known as cache blocking, and merging iterations together to improve reuse across iterations. Cache blocking alone is useful under a very limited set of circumstances [7]. Thus, more contemporary approaches to stencil optimization are geared towards techniques that leverage tiling in both the spatial and temporal dimensions of computation in order to increase cached data reuse [5, 9, 13, 15].

In this paper, we propose to take advantage of these optimizations by building an automatic stencil code generation system.

1.1 Summary of Proposed Work

This proposal outlines the development of an automated system for generating high-quality, platform-adapted implementations of *stencil kernels*. In practice, performance is a complex function of factors such as compiler technology, machine architecture, memory access patterns, instruction scheduling, and prefetching. However, through the use of performance models and heuristics, we expect to find near-optimal algorithms and parameters for a given stencil kernel.

The serial tuning will consist of three distinct, independent sets of optimizations: cache-level optimizations, prefetching optimization, and instruction-level optimizations. Both the prefetching and instruction-level optimizations can be completed offline. However, the cache-level optimizations will require runtime knowledge in order to tune well.

The implementation of this code generator will be done using Sketching [12]. Our experience so far is with hand-optimizations, which are described below. As part of the proposed work we will build a general code generation system for stencil computations, which will support the input of user-defined stencil operators, a variety of boundary conditions, and will generate optimized code to perform one or more sweeps of the stencil computation. During the first phase of this work, we will explore possible code generation frameworks and select one. This exploration will include traditional compiler intermediate forms, such as a control dependence graph [10], or the use of a sketching system that allows the optimizations to be described and validated.

1.2 Hardware Platforms

This proposal examines three leading microprocessor designs to present our stencil optimization results: the Itanium 2, the AMD Opteron, and the IBM Power5. An overview of each platform’s architectural characteristics is shown in Table 1. Other platforms will be included for the actual dissertation.

	Itanium2	Opteron	Power5
Architecture	VLIW	super scalar	super scalar
Frequency (GHz)	1.4	2.2	1.9
Peak (GFlop/s)	5.6	4.4	7.6
DRAM (GB/s)	6.4	5.2	15
FP Registers	128	16	32
(renamed/rotating)	96	88	120
L1 D\$ (KB)	32	64	64
L2 D\$ (KB)	256	1024	1920
L3 D\$ (MB)	3	N/A	36
Introduction	2003	2004	2004
Cores Used	1	1	1
Compiler Used	Intel 9.0	Pathscale	XLC

Table 1: Overview of architectural characteristics.

2 Cache-Level Optimizations

There has been considerable work in memory optimizations for stencil computations, motivated by both the importance of these algorithmic kernels and their poor performance when compared to machine peak. Cache blocking is the standard technique for improving cache reuse, because it reduces the memory bandwidth requirements of an algorithm.

However, the proper utilization of prefetching should not be underestimated. On most platforms, it is the primary strategy for hiding memory latency, and is the reason that the Stanza Triad microbenchmark was created.

2.1 Stanza Triad

In this section we explore prefetching behavior of modern microprocessors using a simple microbenchmark called *Stanza Triad*. The goal of this work is to demonstrate how the requirements for the efficient use of prefetch can compete with, or even interfere with, traditional strategies employed for cache blocking, compromising their effectiveness.

The Stanza Triad (STriad) benchmark is a derivative of the STREAM [8] Triad benchmark. STriad works by performing a DAXPY (Triad) inner loop for a size L stanza before jumping k elements and continuing on to the next L elements, until we reach the end of the array. This allows us to then calculate the bandwidth for a particular stanza length L . The minimum stanza length is always greater than or equal to the cache line size of the microprocessor so that cache-line filling efficiency is not a factor in performance. If we set L to the array length, STriad behaves exactly like STREAM Triad. Pseudocode for the STriad benchmark is shown in Figure 1.

```
while  $i < \text{arraylength}$ 
  for  $L$  elements
     $A_i = \text{scalar} * X_i + Y_i$ 
  skip  $k$  elements
```

Figure 1: STriad pseudocode. In an architecture with no prefetching, the bandwidth should not be dependent on stanza length (as long as $L > \text{linesize}$), since a cache miss will always be incurred at the beginning of each cache line.

Figure 2 shows the performance of the STriad benchmark on three modern microprocessors. It also shows the value from a simple performance model designed to predict memory access overhead for a given stanza length. The model was formulated by first approximating the cost of accessing the first (non-streamed) cache line from main memory, C_{first} , by the overhead of performing an STriad with a short (single cache line) stanza length. C_{stream} , on the other hand, represents the cost of a unit-stride (streamed) cache miss, as computed by performing an STriad where the stanza length is maximized (set to the total array length). The cost of streaming through L words of data, for a given architecture containing W words per cache line, is calculated as $C_{first} + (\lceil L/W \rceil - 1) * C_{stream}$. In other words, we assume that after paying C_{first} to bring in the first cache line from main memory, the remaining data accesses cost C_{stream} due to enabled stream prefetching. Note that this simplified approach does not distinguish between the cost of loads

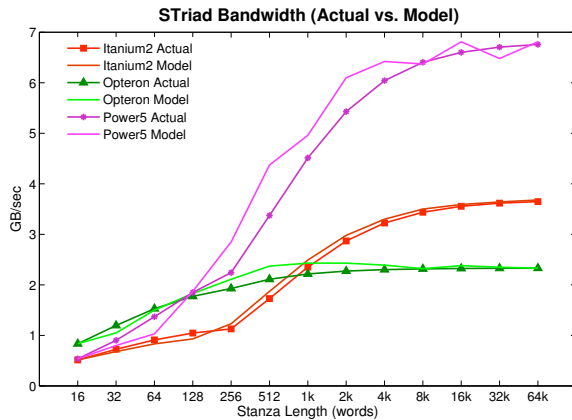


Figure 2: Performance of STriad versus our analytical model on the Itanium2, Opteron, and Power5.

and stores [7].

Our simple two-parameter cost model does a fairly good job predicting performance on the Opteron and Power5. However, the Itanium2 performance was modeled a slightly more complex model since the system utilized two different prefetch mechanisms. By accounting for this, the Itanium2 is modeled fairly well too.

2.2 Single Timestep

Two major obstacles make it difficult to achieve much data reuse within a single iteration. First, there is limited potential for cache blocking in a single stencil sweep. This is because each grid value is used a small, constant number of times, which is simply the number of points in the stencil operator. Second, modern processors contain on-chip caches of sizes relatively large in comparison to main memory size. This means that cache blocking is now effective only for large (and sometimes unrealistic) problem sizes. In two dimensions, Rivera and Tseng [11] have already shown that blocking is unlikely to be effective in practice.

In three dimensions, there is more opportunity for effective cache blocking. However, there is a tradeoff between cache blocking and prefetching. Cache blocking will, of course, reduce memory traffic by reusing data. On the other hand, prefetching will improve performance of stride-1 accesses by hiding memory latency. As a result, cache blocking in the unit-stride dimension will severely disrupt the prefetch engine, and is therefore avoided [7].

2.3 Multiple Timesteps

Performing multiple timesteps affords us the luxury of being able to reuse data across iterations. However, we do make the underlying assumption that no other computation needs to be performed between consecutive sweeps. This may not hold in some cases, but we assume it does here.

2.3.1 The Cache Blocking with Time Skewing Algorithm

The main memory-level algorithm that this proposal will focus on is cache blocking with time skewing [9, 13, 15]. If only one iteration is performed, this algorithm degenerates to the simple cache blocking discussed previously. However, over multiple iterations, each sweep over a given cache block must *skew* from the previous sweep in order to respect dependencies.

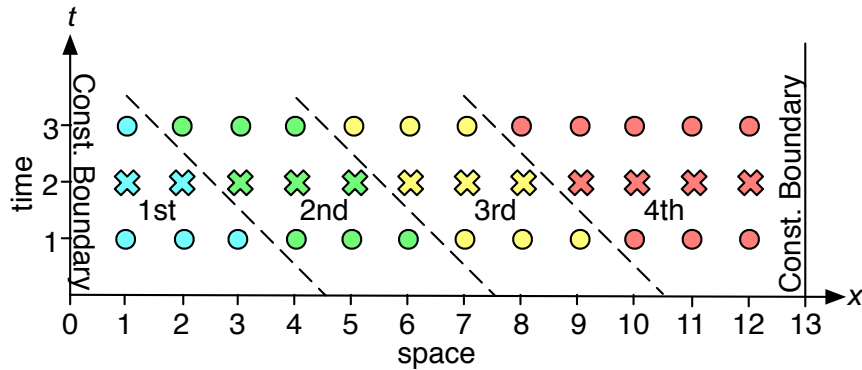


Figure 3: A simplified two-dimensional spacetime diagram of cache blocking with time skewing for a 3-point stencil. The cache blocks need to be executed in the order shown to preserve dependencies. The X's and O's indicate which of two arrays is being written to.

Figure 3 shows a simplified diagram of time skewing for a 3-point stencil. The grid is divided into cache blocks by several skewed cuts. These cuts are skewed in order to preserve the data dependencies of the stencil. For example, the cut between the first and second cache blocks allows the first cache block to be fully calculated before starting on the second cache block. In general, this holds true between the n^{th} and $(n + 1)^{\text{th}}$ cache blocks. As long as the blocks are executed in the proper order, the algorithm respects the stencil dependencies.

A closer representation to our actual 3D time skewing code is illustrated in Figure 4. As stated previously, only two of the three spatial dimensions

are cut to allow for better prefetching in the unit-stride dimension. By showing how the number of stencil operations performed varies within each cache block, the diagram sheds light on how time skewing works in higher dimensions.

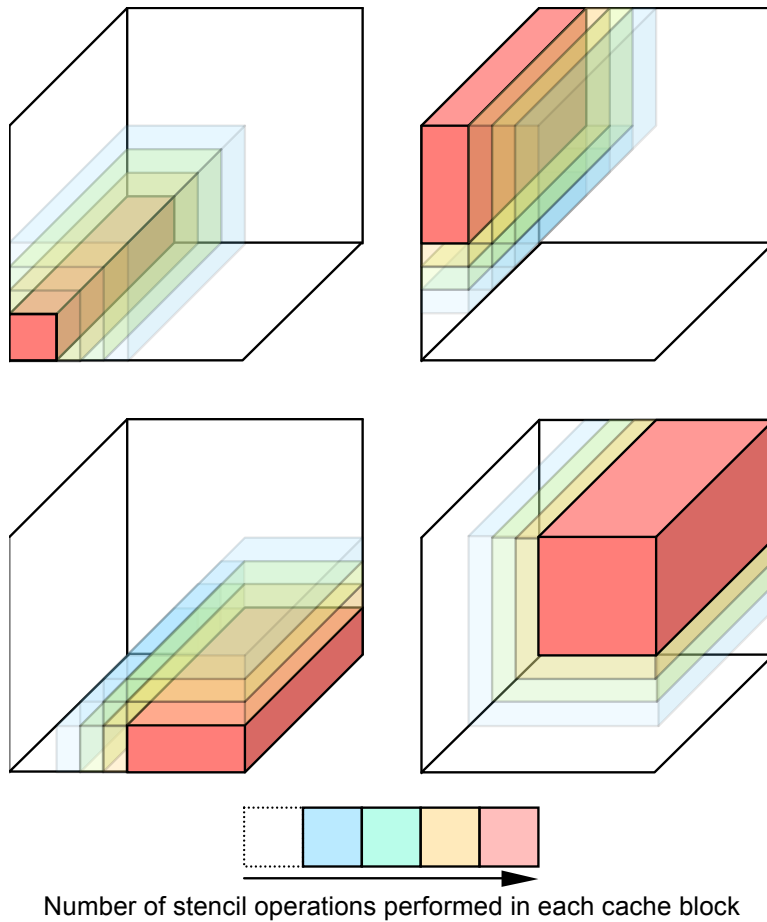


Figure 4: Color coded plots of the number of stencils operations performed on a 10^3 grid using four iteration time skewing with $5 \times 5 \times 10$ cache blocks. There is one plot for each cache block. Blue halos represent only a single stencil operation for that region, where red blocks show the blocks where the full four stencils operations were performed. When processed in order, the full 10^3 has completed four iterations— i.e. a blue cell in four different cache blocks implies one stencil performed in each cache block or four total.

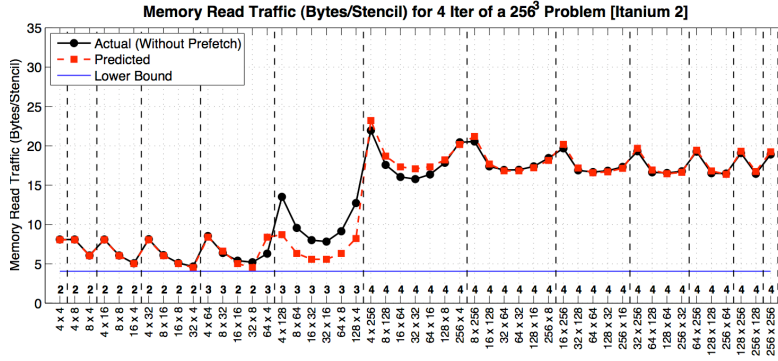
The cache blocking with time skewing algorithm does require the user to specify a cache block size. However, instead of searching over varying cache block sizes, we can create a performance model to help determine the optimal block size. This offers two advantages. First, instead of performing all the runs needed for an exhaustive block search, the performance model only requires some basic machine parameters as input. Second, the model provides greater insight into potential bottlenecks. For instance, by knowing whether the performance is processor or memory-bound, appropriate optimizations can be applied. However, the model does need to be validated before any predictions can be made.

In case the stencil code is memory-bound, we first tried to model the number of cache misses resulting from the different cache block sizes. For each cache block, there are two sources of cache misses: the misses from initially loading it into cache, and the misses from shifting the block (when performing multiple iterations). In the first case, the misses are mostly compulsory. However, the shifting may cause capacity misses because we always shift towards the completed portion of the grid; if the cache is large enough, these misses can be avoided.

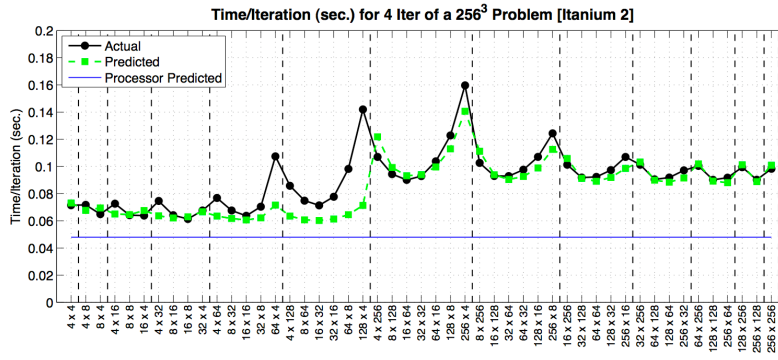
The cache misses were also categorized as being "fast" or "slow". Fast cache misses occur when we are streaming through memory. Since prefetch engines (if present) will retrieve the next cache line in advance, loading the next cache line will typically be a fast operation. Slow cache misses, on the other hand, occur when the next required cache line is not adjacent to the current one. In this case, prefetch engines are usually not useful, and these misses become more expensive. Our model differentiates between slow and fast cache misses because we will use a 2-point STriad microbenchmark to assign different time costs to them. However, in terms of number of cache misses, and consequently memory read traffic, they are equivalent.

In order to model the memory read traffic, each block size was classified as being in one of five distinct cases. Then, we attempted to validate this model against the actual memory read traffic in Figure 5(a). There are two things to note about this figure. First, we did not include prefetching in the memory read traffic model, so in order to make a fair comparison, the software prefetch on the Itanium 2 was deactivated. In addition, conflict misses were included in the performance model as a cumulative Gaussian distribution that matched the data from a microbenchmark.

While the model is not perfect, it does a very good job of predicting the actual memory read traffic for both one and four iterations. However, the figure does not show whether the modeled cache misses were classified as "slow" or "fast". This distinction is reflected in the actual running time,



(a)



(b)

Figure 5: A comparison of the time skewing performance model against reality on the Itanium 2. The graphs show the average (a) main memory read traffic and (b) running times for four sweeps over a 256^3 problem with constant boundaries. On all graphs, the x-axis shows the x and y-dimensions of each plotted cache block size (the z-dimension, which is contiguous in memory, is not shown because it is always uncut). The x-axis is ordered such that the block sizes are monotonically increasing, and the vertical dotted lines divide areas of equal-sized cache blocks.

which is covered in the next section.

However, our ultimate goal is to model performance. For memory-bound applications, we have taken a step in that direction by modeling memory read traffic. However, for processor-bound applications, we need to set up a different model. By combining these two models, we will hopefully be able to predict the actual performance of the stencil code over all cache block sizes.

The first step is to convert the memory read traffic into a running time.

This was done by using the STriad microbenchmark (mentioned in section 2.1) to determine the time for both a slow and fast cache miss. Combining this with our cache miss model, we are able to predict running times for stencil codes that are memory-bound.

The next step is to model processor-bound stencil codes. This was done by running multiple iterations over a small problem that fits into the processor’s L1 cache. During the first iteration, the problem will be loaded into cache, so then all subsequent iterations should be processor-bound. The computation rate for these later iterations should be the maximum we can achieve for this code.

The final step is to reconcile the memory-bound and processor-bound models so that we reasonably predict the running time. This was done in two steps. For the first iteration running time, the memory-bound model was always used, since the problem needs to be loaded into cache. However, for subsequent iterations, the maximum of the two models is used, since that is our bottleneck.

The Itanium 2 results are shown in Figure 5(b). Our model does a fairly good job of predicting overall running time. The model is not as good as the memory traffic model, but we would not expect it to be. The memory traffic model was used in creating the running time model, so any errors present in the memory traffic model would be propagated to the running time model, along with any additional errors in the running time model itself.

2.3.2 Circular Queue Algorithm

Briefly, the circular queue algorithm breaks the grid into equal-sized slabs divided along the y-dimension. Each slab is then executed independently of the others. However, when sweeps are performed over an individual slab, intermediate values are written into *circular queues*. If n sweeps are being performed, $n - 1$ circular queues are maintained, where each circular queue consists of three planes (assuming we use a 7-point 3D stencil). Each circular queue stores only enough data so that a stencil sweep can be performed over the middle plane. Once a given plane is complete, the bottom plane (which is no longer needed) becomes the top plane in the queue and is loaded with new data.

There are several advantages to this algorithm. First, the only parameter that needs to be tuned is the slab size, which constitutes only one dimension. This is in contrast to the time skewing algorithm, which requires two dimensions of the cache block to be specified. Another advantage is that, due to temporal locality, the temporary values in the circular queues will

not be written back to main memory. This would considerably reduce the memory write traffic. Finally, this algorithm is readily parallelizable, since each slab can be given to a different processor to work on independently.

However, this final advantage also has a drawback. The circular queue algorithm performs redundant work for each slab, in contrast to the time skewing algorithm. For large iteration counts, this will become an issue.

2.3.3 Cache Oblivious Algorithm

Another algorithm for performing stencil codes is the cache oblivious algorithm. This algorithm is set up so that it does not require any explicit information about the cache hierarchy. It does this by recursively cutting in the spatial and temporal dimensions until there is only one timestep in the calculation. At this point, the recursion stops and the timestep is performed in the usual manner [5].

This algorithm does do a good job in reducing the overall number of cache misses. However, the prefetch engine cannot be well-utilized because of all the spatial cuts, resulting in poor overall performance. Kamil et al. [6] attempted several additional optimizations to boost performance, but a considerable amount of work was required for incremental improvements in running time.

3 Prefetching Optimization

Another graduate student at UC Berkeley, Sam Williams, recently found that the gcc compiler for the Opteron did not do a very good job of software prefetching. By trying different prefetch distances using the software prefetch intrinsic, he was able to get a 40% improvement in stream performance.

This shows that compilers do not always prefetch well, and is therefore another tunable parameter for our system. This parameter is especially valuable because it made such a large performance improvement on the Opteron. While we should not expect stencil code performance to improve as drastically, we may be able to do better than the compiler-inserted prefetch intrinsics.

4 Instruction-Level Optimizations

Although stencil kernels are often memory-bound during the first iteration, they can be compute-bound for subsequent sweeps. This is why it is important to find a good mix of instructions and schedule them well. There are several instruction-level optimizations that can be attempted to do this, like loop unrolling in various dimensions and array unaliasing/aliasing. However, modeling the performance of these optimizations is extremely difficult, especially when various optimizations are combined. As shown in Figure 6, the performance of instruction-level optimizations can greatly vary from one platform to the next.

As a result, a better way to find the optimal set of instruction-level optimizations is through an exhaustive search. This is not an impractical solution, since the search can be performed on a small problem size—namely, one that fits into the processor’s L1 cache. By executing many sweeps (1000) over this problem, we can confirm that the problem is compute-bound. However, since the problem size is so small, this should still be very fast to execute. Therefore, an exhaustive search over all combinations of instruction-level optimizations should still be tractable.

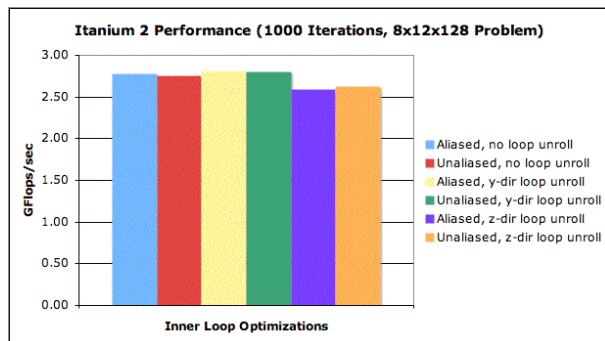
5 Code Generator Proposal

From the previous sections, we have shown that there are many possible optimizations for improving the performance of stencil kernels. Therefore, I propose to build an automatic, platform-adapting stencil code generator that takes advantage of these optimizations.

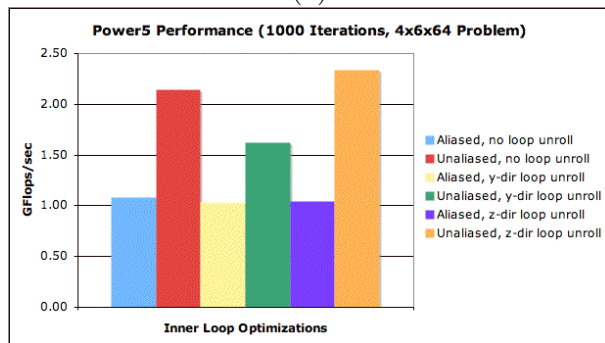
The system will allow the user to specify a particular stencil operator: the number of neighbors involved, the formula used to update a point in the grid, and the dimensionality of the problem. To support various boundary conditions, we will allow the user to specify either constant or periodic boundary conditions.

5.1 Automatic Tuning Libraries

The reason this code generator should automatically tune is to avoid the complications of hand-tuning. Hand-tuning is usually slow, tedious, and error-prone. There is an overhead involved in setting up the infrastructure to tune automatically, but it is certainly worth the cost when considering the large number of platforms used to run scientific codes.



(a)



(b)

Figure 6: Performance of various instruction-level optimizations on the (a) Itanium2 and (b) Power5. By performing many iterations over a problem that fits into L1 cache, we expect the problem to be compute-bound. The same optimizations perform very differently across the two platforms, thereby making a strong argument for using search.

Several common scientific kernels already have automatically tuning libraries, like Atlas [1] for dense matrix multiply, OSKI [14] for sparse matrix multiply, and FFTW [4] for Fast Fourier Transforms. Demmel et al. [3] state that the important design points for these libraries are portability, performance, and ease of use. We shall keep these points in mind as the stencil code generator is being designed.

5.2 Parameter Space Search

From the three different types of stencil optimizations mentioned in this proposal, it is important to note that each of them can be performed independently and offline. This means that at build time, the system can run

experiments to determine the best parameters for each set of optimizations. Once these are complete, no further heuristics or search is required during runtime.

5.3 The Use of Sketching

A. Solar-Lezama et al. [12] have developed a compiler to allow users to try many different stencil implementations easily. First, the programmer gives the compiler a simple reference specification of the desired stencil code. Then, he can provide the compiler with a more complicated stencil implementation that, for instance, is performing an optimized algorithm. The programmer can give the compiler a code implementation with several "holes" in it. This is called a *sketch*. By comparing this sketch to the reference implementation, the compiler is able to determine the correct values for the holes.

Sketching can complement our search-based method of finding optimal code. Instead of having to restrict the space of implementations a priori to make the search tractable, the sketch encodes knowledge that the user has about the best way to implement an algorithm, allowing the search to proceed faster.

6 Proposed Timeline

I plan to complete the research and writing for my thesis in the course of a year and a half. My tentative timeline is as follows:

1. Spring 2007: Complete exploration of the parameter space
2. Summer 2007 – Fall 2007: Explore code generation infrastructure and develop automatic tuning system
3. Spring 2008: Profile and tune an actual stencil-based application (e.g., Chombo, Cactus)
4. Summer 2008: Thesis writing

From the schedule above, I am expecting to write three papers. The first paper is already well underway and talks in detail about various cache-level stencil algorithms and their performance. The second paper would discuss the details and performance of the actual code generator. The final paper would use this system to detail the profiling and tuning of an actual stencil

application. This paper would likely be written in conjunction with my thesis.

7 Conclusion

This proposal describes a system that will generate optimized stencil code for a given platform by taking advantage of memory-level, prefetching, and instruction-level optimizations. Key research contributions of this research include:

- Profiling and addressing the bottlenecks in a common stencil application
- Validating the use of "Sketching" as a viable method for quickly generating complex stencil kernels
- Implementation of an automated, high-performance stencil code generator
- Confirming the usefulness of this system through benchmarking

The ultimate metric of success for this system will be its usage within the scientific and high performance communities.

References

- [1] Atlas Homepage. <http://math-atlas.sourceforge.net>.
- [2] M. Berger and J. Olinger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.
- [3] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, 2005.
- [4] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".
- [5] M. Frigo and V. Strumpen. Evaluation of cache-based superscalar and cache-less vector architectures for scientific computations. In *Proc. of the 19th ACM International Conference on Supercomputing (ICS05)*, Boston, MA, 2005.

- [6] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick. Implicit and explicit optimizations for stencil computations. In *4th Annual ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, San Jose, CA, 2006.
- [7] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *3rd Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago, IL, 2005.
- [8] J. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE TCAA Newsletter*, December 1995.
- [9] J. McCalpin and D. Wonnacott. Time skewing: A value-based approach to optimizing for memory locality. Technical Report DCS-TR-379, Department of Computer Science, Rutgers University, 1999.
- [10] J. Mellor-Crummey and V. Adve. A control-flow simplification algorithm for optimizing compiler-generated parallel code. In *International Journal of Parallel Programming*, volume 26 (5), October 1998.
- [11] G. Rivera and C. Tseng. Tiling optimizations for 3d scientific computations. In *Proceedings of SC'00*, Dallas, TX, November 2000. Supercomputing 2000.
- [12] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 404–415, 2006.
- [13] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, 1999.
- [14] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Proceedings of SciDAC 2005, Journal of Physics: Conference Series*, 2005.
- [15] D. Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. In *IPDPS: International Conference on Parallel and Distributed Computing Systems*, Cancun, Mexico, 2000.