

GPS-TTBP Report

USC-ISI

Jacqueline Chame

jchame@isi.edu

During 2010 we have continued our research and development of compiler technology for automatic performance tuning of high-performance applications. The main activities of this past year were continuing the development of our automatic performance tuning framework and compiler tools and the autotuning of PETSc library functions using specialization.

Autotuning framework and compiler tools.

CHiLL, a framework for composing high-level loop transformations designed to generate efficient code for complex loop nests, developed at ISI and University of Utah, was publicly released in 2010. CHiLL supports an extensive collection of loop transformations for perfect and imperfect loop nests, including tiling, permutation and unroll-and-jam, thus lifting the burden of generating multiple intermediate steps from compilers or optimization tools. CHiLL uses the Omega test to manipulate integer arithmetic and relies on polyhedral scanning provided by Omega's code generator. Recently we have integrated new features into CHiLL, including support for user input to the tool. Users can now relay information about the code and input data that could not be derived by static analysis alone, and this information often results in more efficient code generation. We have also updated the Omega Library, released as Omega library Plus, which is a main component of CHiLL. Both CHiLL and Omega Library Plus have been released and can be found, respectively, at <http://www.cs.utah.edu/~chunchen/chill> and <http://www.cs.utah.edu/~chunchen/omega>.

Over the past year, we have worked extensively to increase the capability of CHiLL. A major focus of this work is on providing the appropriate interface to application and library developers to increase their productivity. In working with application and library developers, we have discovered that many optimizations that are well-known and available in most compilers are performed manually by the application developers. The reason behind this is the application developers do not have sufficient control over how the optimizations are performed or the parameters that are used by the compiler, and they cannot see the results of the optimization. Our goal in building the interface to CHiLL was to expose the transformation capability to the programmer, and then rely on the compiler and auto-tuning framework to automate code generation and parameter selection. Over the past year, we have broadened the capabilities of the interface in CHiLL. We now support OpenMP and CUDA code generation through high-level directives provided by the programmer, and work is underway for OpenCL code generation. We have also raised the level of abstraction for this interface so that programmers have a higher level of abstraction when interacting with the system.

Autotuning of PETSc library functions.

We have developed techniques for combining automatic performance tuning (autotuning) with specialization (in collaboration with Argonne and University of Utah), as reported in 2009. Specialization information allows autotuning tools to derive highly optimized specialized versions of a computation for known input sizes or parameters. CHiLL, a loop transformation framework based on the polyhedral model, can automatically generate specialized versions according to user-specified optimization

strategies. CHiLL provides a high-level script interface that allows a user or compiler to specify an optimization strategy as a sequence of code transformations and parameters.

In the past year we have applied autotuning and specialization to optimize PETSc functions used in many scientific applications (including GTC) such as solvers (MatSolve_SeqBAIJ_N), matrix factorization (MatLUFactorNumeric_SeqBAIJ_N) and matrix multiplication (MatMul_SeqBAIJ_N). These functions use a block sparse matrix representation, where the input matrix is a collection of dense blocks of a fixed size. The N at the end of the function names denotes that the fixed block size is a parameter to the function.

In addition to the size-independent functions, PETSc also provides size-dependent specialized implementations for a range of small sizes, which are hand-optimized. Writing specialized code is a technique often used by library developers to optimize applications. However, manually-written code has several disadvantages when compared to our framework.

- The library developer will not possess the values of parameters along with their frequencies at design time. Hence, he would be able to write specialized functions only for specific values.
- It gets a little difficult for the developer to reason about the best implementation when the number of variables/parameters along with each of their possible values is more than a few.
- It is not feasible to expect the application programmer to write specialized code.
- Performance of implementations varies according to the architecture.

To automate the optimization of the PETSc functions, we start with a clean implementation of the key loop nests, with no pointer arithmetic and eliminating the copies at the beginning of the functions. In this way, we focus on just the computation in the functions. Figure 1 summarizes the autotuning process for triangular solve using our CHiLL transformation and code generation system and the Active Harmony framework from University of Maryland for navigating the optimization search space.

(a) Triangular solver

```
#define SIZE 15
void forward_solve_kernel(...)
{
    ...
    for (cntr=SIZE-1; cntr>=0; cntr--) {
        x[cntr] = t+bs*(*vi++);
        for (j=0; j<bs; j++)
            for (k=0; k<bs; k++)
                s[k]-=v[cntr][bs*j+k]*x[cntr[j]];
    }
}
```

(b) CHiLL transformation script

```
original()
known(bs>14)
known(bs<16)
unroll(stmt1,loop2,u1)
unroll(stmt1,loop3,u2)
```

(c) Constraints on search parameters

```
0 ≤ u1 ≤ 16
0 ≤ u2 ≤ 16
compilers ∈ {gnu, pathscale, cray, pg}
```

(d) Search space

```
17*17*4 = 1156 points
```

Figure 1(a) shows the forward solve portion of the triangular solve code. Figure 1(b) shows a transformation recipe provided to CHiLL to specialize the function for a given block size (15 in the example). The optimization strategy used for this loop nest is to unroll the two innermost loops to expose both instruction-level parallelism and data reuse that can be exploited in registers. The unroll factors are constrained so that they are less than the block size. Figure 1(c) shows the constraints on the possible values for the optimization parameter (unroll factors). In addition, the constraints on the search space of possible code variants specify the set of compilers that will generate the final code.

Active Harmony searches the space of possible unroll factors for this loop nest to identify the one that leads to the best performance. The search space consists of over 1000 points, but Active Harmony employs a parallel rank order search to limit the number of points that must be considered.

Figure 1. Autotuning process for forward solve.

These autotuning experiments were performed on Jaguar, a Cray XT5 at Oak Ridge National Laboratory, and a summary of performance results is shown in Table 1.

Compiler	Original code	Active Harmony search			Exhaustive search		
	Time	Time	(u1,u2)	Speedup	Time	(u1,u2)	Speedup
pathscale	0.58	0.32	(3, 11)	1.81	0.30	(3, 15)	1.93
gnu	0.71	0.47	(5, 13)	1.51	0.46	(5, 7)	1.54
pgi	0.90	0.53	(5, 3)	1.70	0.53	(5, 3)	1.70
cray	1.13	0.70	(15, 5)	1.61	0.69	(15, 15)	1.63

Table 1. Autotuning performance summary for PETSc triangular solve.

The experiment involved trying all combinations of unroll factors for the two innermost loops, and the four compilers available on Jaguar (the four rows) to achieve speedup over the original execution time (the second column). The search space was small enough that it was possible to exhaustively search these options. The exhaustive search results are shown in the rightmost three columns: the execution time, two unroll factors and speedup over the original execution time. By comparison, columns three through five show the comparable results with Active Harmony, showing slightly different unroll factors, and performance. Active Harmony results are obtained by evaluating, on average, less than 50 points, which amounts to less than 5% of the total points in the search space. Another perhaps surprising result is how such a simple piece of code achieves very different performance on the same architecture using different compilers. The optimization strategy and results are similar for the matrix vector multiply. We specialized for different column sizes (105, 90, 75, 60) and row size 15. Since the search space is fairly small, we did an exhaustive parameter sweep (two unroll factors) for the most dominant column sizes 105 and 90 using the PGI compiler. On average, the best CHiLL-generated code variant performs 1.5X faster than the original implementation.

PETSc alone has 29 functions which have been specialized, amounting to a total of 242 manually-written functions. We have been investigating how to combine CHiLL with PETSc such that applications can use the CHiLL framework to generate these specialized versions to reduce the amount of code that is provided by the library and, using auto-tuning and specialization, derive more highly optimized versions of the library functions. This code generation could be deferred until the build of the application so that the code can be specialized for the application and execution context. Such an integration would allow users to study the hotspots in the PETSc library and extract frequent values if possible. The best performing code would then be automatically generated and included in the PETSc library. Applications, such as GTC, that use PETSc library, could take advantage of libraries specialized for the application's context, features and problem sizes.

Autotuning and CO-Design.

In 2010, we collaborated with the GTC team on the proposal “Co-Design Center for Exascale Simulations of Fusion Plasmas” submitted to the Department of Energy.

We believe that auto-tuning technology can play an important role in the co-design of exascale systems and applications. New hardware features and new technology will bring new challenges and opportunities for code generation and optimization. With auto-tuning technology code developers could quickly implement new code variants that take advantage of new designs and technologies and evaluate their impact on the application. As the co-design process converges, new performance critical computations become candidates for optimization. At this point, auto-tuning technology can be used to create domain-specific code generators and libraries of optimized code variants for the exascale version of the application.

Publications (2011 and 2010)

Jacqueline Chame, Chun Chen, Mary Hall, Jeffrey K. Hollingsworth, Kumar Mahinthakumar, Gabriel Marin, Shreyas Ramalingam, Sarat Sreepathi, Vamsi Sripathi and Ananta Tiwari, **PERI Autotuning of PFLOTRAN**, in *SCIDAC 2011*, July 2011.

J. Shin, M. W. Hall, J. Chame, C. Chen, P. F. Fischer, and P. D. Hovland, **Speeding up Nek5000 with Autotuning and Specialization**, in *Proceedings of the 24th International Conference on Supercomputing*, 2010.

G. Rudy, C. Chen, M. W. Hall, M. M. Khan, and J. Chame, **Using a programming language interface to describe GPGPU optimization and code generation**, in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing*, 2010.