Multicore Optimization for Ranger

Jeff Diamond, Byoung-Do Kim, Martin Burtscher, Steve Keckler, Keshav Pingali, and Jim Browne

jdiamond@cs.utexas.edu, bdkim@tacc.utexas.edu, burtscher@ices.utexas.edu, pingali@cs.utexas.edu, skeckler@cs.utexas.edu, browne@cs.utexas.edu

Abstract

As we enter the multicore era, it is of growing concern that an important class of high-performance parallel applications with near perfect weak scaling across nodes cannot take advantage of more than two cores per chip before saturating the on-chip memory hierarchy. This paper presents a first step towards solving this on-chip scalability issue at the source-code level. We begin with a detailed case study of two important simulation benchmarks on the Ranger supercomputing cluster. The two codes are *Homme*, a spectral element code, and *Mangll*, a finite element code. Whereas both applications had previously been classified as a computationally intense and memory light, we instead found both of them to be memory bound, achieving less than 0.5 IPC. Each application has lead to a different strategy for attaining a better memory access/computation balance. Both codes demonstrate substantially worse intra-chip (multicore) scaling than inter-node scaling.

This study is primarily empirical and presents advanced techniques to locate and ameliorate intra-node bottlenecks in applications. As classical optimization for computation is replaced with the more difficult optimization for memory bandwidth, one goal of this study is to make the use of detailed architectural knowledge and low-level program counters more accessible to help a wider programming audience optimize their code. Based on our analysis of *Homme*, we apply several source transformations to reduce memory bandwidth, including utilizing idle FPUs to replace the use of non fundamental memory arrays with computation. The performance characteristics of *Mangll* suggest an optimization strategy for better exploiting the available memory bandwidth by rewriting loops so that the compiler can use vector instructions that load and store multiple data values in a single memory transaction.

1. Problem and Motivation

Attainment of low fractions of peak performance has long been the common situation on modern processor cores. Memory accesses are usually thought to be the culprit limiting performance. The fraction of peak performance attained tends be even lower for multicore chips and multichip nodes of clusters, and their already low ratio of off-chip bandwidth to on-chip FLOPS is expected to continue to decrease in the near future. Hence, we believe an increasing class of simulation codes will ultimately suffer from this bottleneck. For example, intra-node scalability has been shown to be a significant problem on Ranger, where the node configuration is 16 processors per node (four multicore chips). In fact, intra-node performance appears to be the lowest lying bottleneck on this machine. This study aims to take a first step at identifying several heuristics to enable better utilization of on-chip cores.

This paper gives a preliminary report on two systematic case studies of node-level optimization for Ranger. The goal of these studies is to develop guidelines and reusable templates for node-level optimizations. The approach is empirical. The two codes used are *Homme*, a spectral element code that is one of NSF's acceptance benchmark programs, and *Mangll*, a finite element based code modeling tectonic movements of the earth mantle. Both codes are memory bound and exhibit intra-node scalability properties that are significantly different from classically observed inter-node scalability. The study of each code has lead to the investigation of a different approach for improving the computation to memory access ratio.

Section 2 introduces the Ranger supercomputer. Section 3 describes the performance tools we used. Section 4 sketches the experiments conducted on *Homme* and outlines the strategy being developed to increase intra-node performance. Section 5 gives the same coverage to the *Mangll* code. Section 6 provides preliminary conclusions and mentions future research.

2. Ranger

Ranger [9] is a Sun Constellation Linux Cluster at the Texas Advanced Computing Center. It contains 3,936 16-way SMP compute nodes made of 15,744 AMD Opteron processors. In total, the system includes 62,976 compute cores, 123 TB of main memory, and 1.7 PB of global disk space. Ranger has a theoretical peak performance of 579

TFLOPS. All compute nodes are interconnected using InfiniBand in a seven-stage full-CLOS fat-tree topology providing 1 GB/s point-to-point bandwidth.

The quad-core 64-bit AMD Opteron processors are clocked at 2.3 GHz. Each core has a theoretical peak performance of 4 FLOPS/cycle, two 128-bit loads/cycle from the L1 cache, and one 128-bit load/cycle from the L2 cache. This amounts to 9.2 GFLOPS per core, 73.6 GB/s L1 cache bandwidth, and 36.8 GB/s L2 cache bandwidth. The cores are equipped with four 48-bit performance counters and a hardware prefetcher that prefetches directly into the L1 data cache. Each core has separate 64 kB L1 instruction and data caches, both of which are 2-way associative, a unified 512 kB L2 cache that is 8-way associative, and each processor has one 2 MB 32-way associative L3 cache that is shared among the four cores.

3. Performance Tools

In the analysis of large-scale applications in modern HPC environments, utilization of performance tools is essential as the complexity of the system often hinders direct access to critical hardware performance data. While the capability of performance tools is expanding, they are also facing great challenges as the size and complexity of modern HPC systems rapidly increase. With multiple layers of parallel system architecture, a performance tool's ability to process large amounts of profiling data and provide users with a well-designed interface has become very important.

In this study, the Tuning and Analysis Utilities (TAU) [6], HPCtoolkit [7], the PAPI performance counter library [5], and the Pin tool [8] are used for performance profiling. TAU is a well-known profiling and tracing tool that is capable of evaluating most aspects of application performance. It is, however, not as popular as it could be because of its steep learning curve. In contrast, HPCToolkit provides an easy to use interface and a simple profiling process but it is still in the development phase. It works directly with application binaries without requiring instrumentation by the user. It enables transparent access to the PAPI performance counter measurement system, maps the performance data to source code, and presents the results through a graphical user interface that supports top-down analysis. One drawback of HPCToolkit is the lack of a post-processing feature that integrates the performance data from each core. TAU has this feature by default and displays summary results in multiple ways in its graphical viewer. Once the overall profiling is done for the application and major hot spots and bottlenecks have been identified, subroutine or loop-level profiling is necessary for further analysis. TAU requires code instrumentation for this purpose while HPCToolkit is capable of providing detailed information at the loop-level (exclusive and inclusive) without the additional effort of code instrumentation.

In some cases, using PAPI directly was the only way to positively establish what code was being timed and how different performance counters correlate. Direct PAPI was also the easiest way to isolate the performance of specific lines of source code without incurring significant timing overhead. However, using PAPI directly has two main drawbacks. The first is that implementing low-overhead timing of exclusive procedures as well as nested, staggered, and disjoint intervals requires the development of custom timing code on top of PAPI. The second productivity issue with direct PAPI timing is the need to change and recompile the source code for different timing configurations and managing collections of executables that provide different measurements. However, if source code changes are being made as a part of the optimization process anyhow, then the need to modify and recompile source code for timing is less of an issue. Moreover, once sophisticated PAPI libraries are developed, they can be amortized over many optimization projects.

Pin does not require recompilation of the source code, but it does require programming specific timing analysis procedures as separate shared libraries, which need to be linked with the executable as part of a Pin timing run. Also, the amount of data generated by Pin can be so substantial that significant live processing is needed to analyze data on the fly. Finally, correlating source code addresses with Pin addresses requires access to debugging information or live code address information and can be tedious.

We used both Pin and direct PAPI to provide extremely detailed performance analysis of program code as well as to correlate performance statistics with those of high-level binary instrumentation tools like TAU and HPCtoolkit. Overall, the use of direct PAPI calls and Pin were extremely valuable, but the much lower productivity rate observed demonstrates the value and need of high-level binary instrumentation tools. We hope that in the future these tools become more accurate and provide greater control over their overhead so that hand-coded tools are not needed.

4. Analysis of the *Homme* Code

4.1 Description of *Homme*

Homme (High Order Method Modeling Environment) is an atmospheric general circulation model (AGCM) consisting of a dynamical core based on the hydrostatic equations, coupled to sub-grid scale models of physical processes [4]. The *Homme* code is designed to provide 3D global atmospheric simulation similar to the Community Atmospheric Model (CAM), which is a subcomponent of the second generation Community Climate System Model (CCSM-2). The code is based on 2D spectral elements in curvilinear coordinates on a cubed-sphere combined with a second order finite difference scheme for the vertical discretization and advection [2].

The benchmark version of *Homme* was one of NSF's acceptance benchmark programs for Ranger. It solves a modified form of the hydrostatic primitive equations with analytically specified initial conditions in the form of a baroclinically unstable mid-latitude jet for a period of twelve days, following an initial perturbation [3]. Whereas the code is designed for using hybrid parallel runs (both MPI and OpenMP), the benchmark version uses MPI-only parallelism. Although a semi-implicit scheme is used for time integration, the benchmark version of *Homme* is greatly simplified and spends most of its time in explicit finite difference computation on a static regular grid.

4.2 Performance Characteristics of *Homme*

Homme was chosen for its archetypal use of piece-wise regular data structures and concentration of performance critical code. We confirmed that eleven procedures account for 90% of the execution time, with about two thirds of the execution time occurring in simple finite difference code. All computation takes place on small independent blocks of 8x8x96 grid elements, with about 17% of the execution time spent in exchanging boundary data between four nearest neighbor elements.

Homme is a classic example of a full scale program in which cursory performance data can actually be quite misleading. Internode scalability studies of *Homme* show near perfect weak scaling from 16 to 4096 cores while keeping the number of active cores per node, or "core density", constant. (We run one single-threaded process per "active" core.) The communication overhead remains at a constant 20% of the execution time, even as the work per core is varied from 3 to 800 elements. An investigation of the main computation step in *Homme* reveals that over half of all instructions are floating point, and, most importantly, the L1 cache miss ratio averages just 3% and shows negligible variation with core density. A source code analysis showed the use of advanced programming techniques: all computation is performed on small, independent blocks of 8x8x96 elements. The loop iteration order maximizes sequential access to data, loops are unrolled by hand, and explicit temporaries maximize register usage. A detailed memory address trace analysis revealed that 95% of all memory accesses are within 16 elements of each other and exhibit full cache line utilization, giving *Homme* a memory access pattern close to that of a perfect streaming application. All these facts would seem to justify *Homme*'s historic classification as computationally heavy, memory light.

However, when factoring in the actual cycle count, a very different picture emerges. Our analysis of the main computation using 4 active cores per chip revealed an average of just 1 instruction executed every 2 cycles and 1 FP operation every 4 cycles, representing a computational efficiency of just 12% compared to the theoretical peak performance without vector instructions. Almost the entire remaining half of the instructions are loads and stores, yet the computation only averages one L1 load every 4 cycles – again only about 12% of non-vectorized load/store efficiency.

A key experiment to help understand the source of this poor performance was to look at intra-node scaling – varying the number of active cores per chip and comparing the performance. Traditionally, this is done by utilizing more cores per chip, and then defining the efficiency as the fraction of linear speedup obtained from increasing the total core count. However, to maximally isolate the performance effects of sharing chip resources between cores from the effects of scaling the problem size and communication patterns, we keep the total number of cores (and hence the work per core) constant and simply vary the number of active cores per chip. We call this a "core density study", as the only changes in program execution are the number of chips and active cores per chip. If a program had perfect intra-node scaling, then total execution time would not change as core density is varied. In fact, the performance might increase with core density since less chip to chip and node to node communication is needed.

The measured intra-node scaling on *Homme* is very poor. Varying the core density while keeping the workload identical reduces the program's execution time to 89% at two cores per chip and to 65% at four cores per chip, despite *Homme* being nearly embarrassingly parallel. Again, keeping the workload and active core count constant but dropping the active core density from 4 to 2 cores per chip increased completion time, FLOP rate, and data bandwidth (consumed GB/sec/core) by 40% while leaving the L1 and L2 miss ratios almost unchanged. Dropping the

active core density from 4 to 1 core per chip increases completion time, FLOP rate, and data bandwidth by 60%. This efficiency drop is even more surprising considering that a lower core density requires more inter-node communication over the network. Note that a more traditional scaling test that doubles the total number of cores when doubling the core density with a constant problem size still yields some net performance benefit with additional cores per chip, but the performance increase is extremely slight, and there is an order of magnitude less performance benefit than when using more chips instead of more cores. Clearly, memory bandwidth is an issue for *Homme*, and a low L1 miss ratio is not an indication of being computationally bound. However, the lack of any significant change in L1 and L2 cache miss rates as the core density increases indicates that the primary issue is off-chip bandwidth per core, and not the L3 cache capacity per core. Since cache capacity is not an issue, we do not have to worry about memory access patterns that artificially reduce cache capacity, e.g., cache conflict misses due to low associativity.

4.3 Balancing Memory Accesses and Computation

We profiled *Homme* using gprof to obtain the percentage of time spent exclusively in each procedure (not including the time of procedure calls made by that procedure.) The rankings of the top 11 procedures varies somewhat depending on the compiler used (PGI vs. Intel) and the active core density (1 to 4 active cores per chip), but there is little difference qualitatively. Half the runtime is spent as follows: about 1/3rd of the execution time is spent in a procedure that computes an explicit finite difference and nearly 40 different fluid dynamics properties, about 17% of the execution time is spent on packing and unpacking element boundaries, and about 11% of the execution time is spent in a diffusion procedure. Adding in gradient (8%), divergence (4%), and vorticity (3%) operators along with preq_robert time integration (6%) accounts for 82% of the total program execution time. All these procedures are memory bound, averaging at most one multiply-add per memory operation, leaving the computational resources of the cores highly underutilized. Source analysis of these procedures reveals a variety of computational styles or "templates" that necessitate different strategies for bandwidth optimization. Comparing exclusive procedure execution times at different active core densities, e.g., 1 core per chip and 4 cores per chip, highlights which procedures are the most affected by bandwidth limitations and where the most improvement can be made. A brief description of these procedures follows.

The procedure exclusively responsible for 1/3rd of the execution time is a large (~800-line) procedure consisting of many compact initialization loops. This procedure precomputes many different fluid dynamic properties of the grid, storing each property in its own array and consuming an enormous amount of bandwidth. In many cases, given properties only differ from the fundamental fluid dynamic properties (temperature, velocity, and log pressure) by a constant scale. In such cases, it may be beneficial to replace the use of these arrays by the computation itself. Direct optimization of this procedure is difficult, as it simply creates data for the rest of *Homme* to use, requiring intimate knowledge of the entire program. This procedure additionally calls 9 of the remaining 10 most performance critical procedures.

The second most performance critical procedure, the diffusion step, takes up 11% of the execution time and calls 3 other performance critical procedures. This procedure is self contained enough that direct optimization is feasible. The two primary approaches that show promise for cutting bandwidth in half are reuse of temporary arrays and replacing temporary arrays by computation. This procedure uses separate local temporary arrays for every element processed by that core, instead of reusing one element's temporaries. Additionally, up to 5 temporary arrays could be replaced by direct computation to determine update values for the temperature and velocity. Such optimizations are complicated by calls to other procedures, which would need to be reordered. Another key to making efficient use of replacing memory utilization with computation is separaing the temperature and velocity computations, which are currently interleaved throughout the code.

The preq_robert time integration (6%) is a pure streaming procedure similar to DAXPY – no data is reused. Without altering the simulation pattern itself, optimization would depend on utilizing huge pages and maximizing access patterns to maximize the use of DRAM banks.

Packing and unpacking element boundaries combined accounts for 18% of the execution time. The primary approach to optimization is to reduce the amount of packing and unpacking and better controlling when it occurs. Another approach is to utilize the fact that a single core will contain a large number of connected elements and logically operate on a larger grid.

The remaining procedures, gradient, divergence, vorticity (which combined account for 15% of the execution time) are classic stencil templates. Due to their access patterns, they are not good candidates for bandwidth reduction. With stencils and packing, it may pay to compute the element edges separately from the core.

5. Analysis of the *Mangll* Code

5.1 Description of *Mangll*

Mangll [1] is a scalable adaptive high-order discretization library used, for example, in simulations of convection in the Earth's mantle and of global seismic wave propagation. It supports dynamic parallel adaptive mesh refinement and coarsening (AMR, through an interface to the p4est library), which is essential for numerical solution of the partial differential equations (PDEs) arising in many multiscale physical problems. *Mangll* provides nodal finite elements on domains that are covered by a distributed hexahedral adaptive mesh with 2:1 split faces and implements the associated interpolation and parallel communication operations on the discretized fields.

We use the *Mangll* code for the numerical solution of the energy equation that is part of the coupled system of PDEs arising in mantle convection simulations, describing the viscous flow and temperature distribution in the lower and upper mantle (the region between the Earth's crust and outer core, spanning depths between about 100 km and 2,870 km). The discretization of this spherical shell yields a large unstructured mesh composed of hexahedral elements that are refined and coarsened on the fly to track the highly localized features of flow and temperature fields. In the mesh adaptation process, a 2:1 size condition between adjacent element faces is maintained.

The energy equation is discretized by a nodal discontinuous Galerkin (DG) ansatz using tensor-product Lagrangian shape functions of arbitrary polynomial order. The spherical shell geometry is realized by an analytical transformation that maps a conforming collection of appropriately connected adaptively subdivided octrees to the computational domain, ensuring a uniform aspect ratio of all elements. The mesh is dynamically repartitioned at each adaptation to guarantee load balance. The time integration is performed with a Runge-Kutta (RK) method. Thus, the mathematical kernel consists of the evaluation of the right hand side for the RK method, which involves the application of the spatial operators in DG discretization, such as tensor derivative matrices, inter-element jump terms and boundary conditions. The main part of the computational work is spent in the volume terms, dominating the surface terms. Additionally, the 2:1 adaptive mesh requires interpolation and lifting operations between neighboring elements on both the same and adjacent processors. The *Mangll* library has been weakly scaled to 32,768 cores on Ranger, delivering a sustained performance of 145 TFLOPS.

5.2 Performance Characteristics of *Mangll*

The *Mangll* code is dominated by two key procedures containing several important loops that perform a large number of small dense matrix-vector operations. Together, the two procedures account for over 50% of the total runtime. One of these procedures has complex memory access patterns within its loops involving indirect array accesses whereas the other procedure only uses strided accesses in its loops. Even though each key loop touches hundreds of megabytes of data, they all have L1 data-cache miss ratios under 1.2% because of the hardware prefetcher. Both procedures execute fewer than two compute instructions per memory access on average. Neither the Intel nor the PGI compiler manages to vectorize the memory accesses in any of these loops. The complex loops execute only one instruction per 3 to 4 machine cycles (i.e., the CPI is 3 to 4) and the simple loops take close to two cycles to execute a single instruction. Given the frequent memory accesses and the low L1 cache miss ratios, the culprit for these poor CPI numbers must be the L1 load-to-use hit latency of three cycles, which cannot be (fully) hidden because there are not enough independent instructions available to execute. In other words, *Mangll* is a memory bound application and the primary performance bottleneck is accesses to the L1 data cache.

5.3 Sketch of Strategy for Vectorizing Memory Accesses

The approach to optimization is to reduce the number of memory accesses by rewriting the loops so that the compiler can use vector instructions, which load and store multiple data values in a single memory transaction. All important loops in the *Mangll* code are either not vectorized at all or only partially vectorized. In the partially vectorized loops, some or all of the computations are vectorized but none of the memory accesses. However, in a memory-bound application with a low L1 data-cache miss ratio such as *Mangll*, it is paramount to vectorize the memory accesses. Only vectorizing the computations does not improve the performance (much) as the computations are not on the critical path, i.e., they are hidden by the memory access latency whether they are vectorized or not. Hence, the goal of this section is to investigate what source-code modifications are necessary to enable the Intel compiler on Ranger to fully vectorize important loops.

We demonstrate the necessary source-code changes on one of *Mangll*'s key loops. The loop and some surrounding context are shown below. It is the most time intensive loop in *Mangll* that can be vectorized. There are a couple of more important loops, but they contain indirect array accesses, which prevents vectorization. The shaded loop in

Figure 1 represents 4.3% of the total runtime. There is a second, almost identical copy of this loop in the code, which takes the same time to execute and all the optimizations described below apply equally to this second loop. More importantly, we believe the general strategy outlined herein is applicable to a broad range of time-intensive loops in simulation codes.

The code shown in Figure 1 is the original *Mangll* code except for a common subexpression that was factored out for readability (**tmp**). The procedure containing this loop is called several times during the course of the execution. Because the measured run uses fourth-order function approximations, the number of inner loop iterations defined by **MMM** is always 125. The number of outer loop iterations, defined by **K**, is roughly 193,000 but varies by a few percent as the mesh adapts. Accesses to the **Vd** array touch 3 kB of data for every loop execution, but the array is reused between different loop invocations and should therefore be resident in the L1 data cache. Accesses to **rhsud** and especially **gd** stream through memory sequentially and touch a total of roughly 2 GB of data for each invocation of the outer **jb** loop. While these accesses have no temporal locality and the amount of data far exceeds all cache sizes in the system, they exhibit perfect spatial locality and follow simple access patterns, which the prefetcher can easily handle. Because the prefetcher prefetches directly into the L1 cache, the L1 data cache miss ratio is under 0.8% for this loop in spite of the large amount of data touched.

```
const int K = ...;
const int MMM = ...;
int ib, jb, M3jb, tmp;
const double *restrict gd = ...;
double ur, us, ut, ux, uy, uz;
double *restrict rhsud = ...;
double *restrict Vd;

for (jb = 0; jb < K; ++jb) {
  compute (Vd);
  compute (Vd + MMM);
  compute (Vd + 2 * MMM);
  ...

  for (ib = 0, M3jb = MMM * jb, tmp = 9 * M3jb; ib < MMM; ++M3jb, ++ib, tmp += 9) {
    ur = Vd[ib];
    us = Vd[ib + MMM];
    ut = Vd[ib + 2 * MMM];

    ux = ur * gd[tmp    ] + us * gd[tmp + 1] + ut * gd[tmp + 2];
    uy = ur * gd[tmp + 3] + us * gd[tmp + 4] + ut * gd[tmp + 5];
    uz = ur * gd[tmp + 6] + us * gd[tmp + 7] + ut * gd[tmp + 8];

    rhsud[M3jb] += ux * ux + uy * uy + uz * uz;
  }
}
```

Figure 1: Key *Mangll* loop with poor vectorization.

This loop is not vectorized for two reasons. First, the array accesses stride through memory at different rates. Second, because **MMM** can be (and in our example is) an odd number, one or two of the three **Vd** components and **rhsud** in every other iteration start at a non-16-byte-aligned memory address.

Improving the **Vd** accesses is simple because the three components are independent, small, reused, and local to this procedure. Hence, they can trivially be replaced by three local arrays (called **ta**, **tb**, and **tc**) that are guaranteed to be 16-byte aligned.

gd stores nine independent vectors in an interleaved fashion because having nine separate vectors (in addition to the "three" **Vd** vectors and the **rhsud** vector) accessed in the same loop would likely result in conflict misses in the two-way associative L1 data cache and hurt performance. However, by allocating a few extra doubles in these vectors and intelligently adjusting the pointers to not necessarily point to the first element, it can be guaranteed that none of the nine arrays start at addresses that map to the same line in the L1 cache, thus ensuring that there will be no conflict misses when streaming through them. With this adjustment, **gd** can safely be separated out into nine independent vectors (called **t0** through **t8**), each of which is accessed with unit strides.

Unfortunately, these nine new vectors as well as **rhsud** will be accessed starting at a non-16-byte-aligned memory address in every other iteration because **MMM** is odd. To avoid this problem, it is necessary to increase the length of each of these vectors by a factor of (**MMM**+1)/**MMM**, i.e., to insert a dummy element at every (**MMM**+1st) position and increase the loop count by one to make it even. This optimization, which trades off a little extra memory (less than 1% in our case) for improved vector performance, has the added benefit of keeping the compiler from duplicating the loop body to handle the first or last iteration in non-vector mode, which may hurt the instruction-cache performance.

The shaded loop in the improved code shown in Figure 2 computes the same result as the shaded loop in Figure 1 but is fully vectorized by the compiler, including the load and store instructions. Note that the code is somewhat idealistic and primarily meant to highlight what changes are necessary to achieve full vectorization. In particular, breaking **gd** up into nine vectors and inserting dummy elements into them and the **rhsud** vector require global changes to the *Mangll* code. We are working on including these changes but have not yet completed this task. Hence, for the time being, we just copy the **gd** data into local arrays before the loop and copy the result into **rhsud** after the loop. Of course, this copying is just overhead and should be avoided. Nevertheless, even with these copy instructions, we observe a small speedup.

```
__declspec(align(16)) double t0[126], t1[126], t2[126], t3[126], t4[126], t5[126];
__declspec(align(16)) double t6[126], t7[126], t8[126], t[126], ta[126], tb[126], tc[126];

copy data to t0, t1, etc.

for (ib = 0; ib < 126; ++ib) {
  ur = ta[ib];
  us = tb[ib];
  ut = tc[ib];

  ux = ur * t0[ib] + us * t1[ib] + ut * t2[ib];
  uy = ur * t3[ib] + us * t4[ib] + ut * t5[ib];
  uz = ur * t6[ib] + us * t7[ib] + ut * t8[ib];

  t[ib] = ux * ux + uy * uy + uz * uz;
}
for (ib = 0; ib < MMM; ++ib) {
  rhsud[M3jb + ib] += t[ib];
}
M3jb += MMM;
```

Figure 2: Optimized *Mangll* loop with full memory vectorization.

All floating-point operations and memory accesses are performed exclusively with SSE vector instructions in the shaded improved loop. Moreover, no startup, cleanup, non-vectorized, or partially vectorized code is emitted, as was verified by a detailed study of the generated assembly code.

We hardcoded the loop count to 126 for two reasons. First, we had to make it even to avoid the need for a non-vectorized iteration at the beginning or the end of the loop. Second, we had to make the loop bound different from the bound of the following loop. As it turns out, the compiler first tries to fuse loops and then vectorizes the code. Hence, it would fuse the not-fully-vectorizable code from the second loop with the first loop, thus preventing several instructions in the first loop from being vectorized.

The **__declspec(align(16))** specification aligns data to a 16-byte boundary. This is necessary to allow the (Intel) compiler to generate vector load and store instructions. Note that when pointers are used, the compiler sometimes emits two versions of the code, one with and one without vectorized memory accesses, as well as a run-time test to check whether the pointed to address is 16-byte aligned so that the vectorized version of the code can be used. Hence, it is important to ensure that dynamically allocated vectors also be 16-byte aligned.

Comparing the old and new loop implementations, we find that the number of executed instructions is 1.8 times lower and the number of L1 data cache accesses is 1.5 times lower due to the vectorization. There are 1.2 times fewer L1 misses, but we suspect that copying the data into the local arrays just before the loop is responsible, as the old and the new loop touch the same amount of data in almost the same access pattern. The L1 data-cache miss ratio is under 1% in both implementations. Finally, based on the cycle count, the new loop executes 4.9 times faster than the old loop, but we attribute any speedup beyond the above factors of 1.5 or 1.8 to the (artificial) reduction in cache

misses. Nevertheless, we believe vectorization to be very important, especially of load and store instructions in memory-bound code, because it improves the memory throughput (by up to a factor of 1.5 in our case).

6. Conclusions and Future Research

The preliminary results from our experimental studies on two representative HPC codes have led to hopefully reusable patterns for restructuring and/or redesigning HPC codes for better intra-node performance. Future research will focus on the application and evaluation of the restructurings that we have suggested as well as seeking further mechanisms for intra-node performance enhancement. The ultimate goal is to package the measurement and restructuring processes so that they can easily be routinely applied on Ranger.

7. Acknowledgements

This research was supported by the National Science Foundation under a grant from the Office of CyberInfrastructure. We are very grateful to Carsten Burstedde, Omar Ghattas, Georg Stadler, and Lucas Wilcox for providing the *Mangll* code and helping us understand and analyze it.

8. References

[1] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong, "Scalable Adaptive Mantle Convection Simulation on Petascale Supercomputers", *In Proceedings of SC'08*, 2008.

[2] R. D. Loft, S. J. Thomas, and J. M. Dennis, "Terascale Spectral Element Dynamical Core for Atmospheric General Circulation Models", *In Proceedings of SC'01*, 2001.

[3] L. M. Polvani, R. K. Scott, and S. J. Thomas, "Numerically Converged Solutions of the Global Primitive Equations for Testing the Dynamical Core of Atmospheric GCMs", *American Meteorological Society, Vol. 132, No. 11, pp. 2539-2552*, 2004.

[4] S. J. Thomas and R. D. Loft, "The NCAR Spectral Element Climate Dynamical Core: Semi-Implicit Eulerian Formulation", *Journal of Scientific Computing, Vol. 25, No. 1/2, November 2005*.

[5] http://icl.cs.utk.edu/papi/

[6] http://www.cs.uoregon.edu/research/tau/home.php

[7] http://www.hpctoolkit.org/

[8] http://www.pintool.org/

[9] http://www.tacc.utexas.edu/resources/hpcsystems/#constellation