OPTIMIZATIONS ON FINITE THREE DIMENSIONAL LARGE EDDY SIMULATIONS

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of the Ohio State University

By

Nandan Phadke, B.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2015

Master’s Examination Committee:

Dr. P. Sadayappan, Advisor

Dr. Louis-Noël Pouchet
Abstract

Large Eddy Simulation kernels working on 3 dimensional point grids using an Alternate Direction Implicit (ADI) scheme for solvers face the challenge of having a significant portion of the computation access non contiguous data, resulting in decreased performance. We focus on this issue with FDL3DI, an implementation of LES, and suggest potential areas of optimization. We pursue a data layout transformation as a solution to this problem and evaluate its effectiveness.

We also identify the coarse grained nature of any parallelism employed in the kernel and attempt to increase performance of the overall kernel by utilizing fine grained parallelism using the many-core architecture of the Intel Xeon Phi accelerator. We develop a hybrid OpenMP+MPI framework for this microkernel and optimize it for Xeon Phi using a variety of techniques. We compare performance characteristics on the Xeon Phi with that on a traditional CPU for a range of optimizations and problem sizes. We evaluate the use of offload mode and quantify the data transfer costs that limit performance and assess the potential for effective use of such accelerators.
Dedication

This thesis is dedicated to my parents and my sister
Acknowledgements

I would like to convey my gratitude to my advisor and mentor Prof. P. Sadayappan and would like to thank him for the opportunity to be a part of this vibrant and enthusiastic research group. He has been a continuous source of inspiration in my entire masters program and I have learnt a great deal in High Performance Computing from him. I would also like to thank Prof. Datta Gaitonde. This work would have not been possible without his continuous support and encouragement. I thank Prof. Louis-Noel Pouchet for his valuable and insightful feedback and suggestions to this work.

I thank my lab seniors John, Venmugil, Martin, Samyam, Kevin for guiding me and providing insights. I also acknowledge the support from my peers Abhijeet, Shashank and Wenlei and thank them for all their help.

I thank my dear friends Sameer Kanase, Neeraj Joshi, Cajetan D'Silva, Gaurav Shah, Jigar Bhanushali, Tejaswi Konduri, Omkar Bahirat for their continuous support.

I would like to thank my relatives Amolika Bhole, Sharmila Deshpande, Chetan Kale and Manjalee Shastri for their encouragement of my endeavours.

Lastly, I would like to thank my parents and my sister for their constant support.
Vita

June 2009 – May 2013 …………… B.E. in Computer Engineering, University of Pune, Pune, India.

Jan 2014 – May 2015 …………… Graduate Research Associate, High Performance Computing Lab, The Ohio State University, Columbus, Ohio, USA

June 2014 – May 2015 …………… Software Development Engineering Intern, MIM Software, Cleveland, Ohio, USA

Fields of Study

Major Field: Computer Science and Engineering

Specialization: High Performance Computing
Table of Contents

Abstract ........................................................................................................................... ii

Dedication ....................................................................................................................... iii

Acknowledgements .......................................................................................................... iv

Vita ..................................................................................................................................... v

Table of Contents ............................................................................................................. vi

List of Tables .................................................................................................................... ix

List of Figures ................................................................................................................... x

Chapter 1: Introduction ................................................................................................. 1

Chapter 2: Problem Statement ......................................................................................... 5

  2.1 Data Layout Transformations ................................................................................ 5
      2.1.1 ADI Scheme ..................................................................................................... 5
      2.1.2 Performance bottlenecks with the ADI scheme .......................................... 6
      2.1.3 Problems with non unit stride access ............................................................ 7

  2.2 Fine grained parallelism and use of Xeon Phi ................................................... 8
      2.2.1 Potential for fine grained parallelism ............................................................... 8
      2.2.2 Xeon Phi ......................................................................................................... 9
      2.2.3 Exploiting parallelism with Xeon Phi ............................................................. 10
Chapter 3: Design

3.1 Data Layout Transformations

3.1.1 Program Structure
3.1.2 Data Reorganization
3.1.3 Loop Permutation and unrolling
3.1.4 Implementation issues

3.2 Optimizing on Xeon Phi

3.2.1 Vectorization
3.2.2 SMP parallelism with OpenMP
3.2.3 Offload from host to MIC
3.2.4 Challenges in implementation for Xeon Phi

Chapter 4: Experimental Results

4.1 Experimental Setup

4.1.1 Oakley
4.1.2 Hydra

4.2 Data Layout Transformations

4.2.1 Data Reorganization
4.2.2 Loop Permutation with Unrolling

4.3 Optimizing on Xeon Phi

4.3.1 Vectorization
4.3.2 OpenMP
4.3.3 Offload from host to MIC
4.3.4 Improving offload performance

Chapter 5: Conclusion
Bibliography .......................................................................................................................... 40
List of Tables

Table 1: Vector Instruction Sets - Vector Lengths ......................................................... 8
Table 2: Data reorganization - Application Execution Time & Speedup ......................... 24
Table 3: Data reorganization - Kernel Performance & Speedup .................................. 25
Table 4: Unroll factor 4 - Application Execution Time ................................................ 26
Table 5: Unroll factor 8 - Application Execution Time ................................................ 26
Table 6: Unroll factor 16 - Application Execution Time ............................................... 27
Table 7: Unroll factor 16 - Kernel Performance Data .................................................. 27
Table 8: Unroll factor 32 - Application Execution Time ............................................... 28
Table 9: Unroll factor 32 - Kernel Performance Data .................................................. 28
Table 10: Vectorization Execution Time - Oakley ......................................................... 29
Table 11: Vectorization Execution Time - Xeon Phi ..................................................... 29
Table 12: OpenMP Execution Time - Xeon Phi ............................................................ 30
Table 13: MIC Offload Performance ............................................................................. 35
Table 14: Offload Bandwidth ......................................................................................... 36
Table 15: MIC Compute vs Oakley 1 Node speedup ..................................................... 37
List of Figures

Figure 1: Process work division ................................................................. 2
Figure 2: Grid along I .................................................................................. 3
Figure 3: Grid along J .................................................................................. 3
Figure 4: Grid along K .................................................................................. 3
Figure 5: Data structure with both read & write accesses ......................... 18
Figure 6: False Sharing ............................................................................... 19
Figure 7: OpenMP Static Schedule allocation ............................................. 20
Figure 8: Performance variation on MIC- 102400 Problem Size ................ 32
Figure 9: Performance variation on MIC- 1024000 Problem Size .............. 33
Figure 10: Performance variation on MIC- 10240000 Problem Size .......... 34
Figure 11: MIC Offload Time Breakup ....................................................... 36
Chapter 1: Introduction

As the design of fluid dynamic systems becomes increasingly complex, it becomes progressively critical to accurately model the turbulence of complex geometric flows. Such simulations lead to better analysis of these systems. But accurate modeling of these systems using direct numerical simulation is exceedingly expensive and is almost impossible even on today’s fastest supercomputers. The popular approach is to use Large Eddy Simulations [1]. Operating on the Navier-Stokes equations, LES attempts to reduce the computation costs of these simulation models, enabling very good accuracy and acceptable execution time.

In this thesis we focus on one such implementation of a three dimensional LES in the FDL3DI code [2], which is based on the general form of the equation:

\[
\frac{\partial \hat{X}}{\partial t} + \frac{\partial \hat{F}_i}{\partial \xi} + \frac{\partial \hat{G}_i}{\partial \eta} + \frac{\partial \hat{H}_i}{\partial \zeta} = \frac{\partial \hat{F}_\nu}{\partial \xi} + \frac{\partial \hat{G}_\nu}{\partial \eta} + \frac{\partial \hat{H}_\nu}{\partial \zeta} + S
\]

The point grid, which we simulate, is a 3 dimensional grid and during the computation we divide up this master problem space into separate 3D grids mapped onto a virtual 3D grid of processors. Let X, Y, Z be the dimensions of this processor grid. The set of X*Y*Z 3D points on the grid are distributed among processes using a block distribution, as illustrated below. Let us consider the example where we run the
simulation on 125 processors and the processor grid is thought of as a \((5 \times 5 \times 5)\) grid. As shown in the figure below the input block is divided into \(5 \times 5 \times 5 = 125\) smaller blocks where one block is given to each processor.

![Figure 1: Process work division](image)

Thus at this point each processor concurrently computes on a sub-system of equations over the smaller grid assigned to it. Along with this, in order to propagate the effect of local computation over the rest of the input grid, boundary ghost cells are periodically communicated across processor boundaries. This reduces communication costs without impact on the computational integrity and helps us converge on to the solution faster. Moreover the 3-D grid at each processor is divided along 3 distinct types of 2-D grids along each dimension and computes the equations along each of the 3 types
of grids. This alternate direction implicit approach is common among many 3D compute kernels and is particularly effective. The division of the 2-D grids is as shown in figures shown below.

Although this general approach is extremely useful in the sense that it enables us to successfully compute a simulation that was impossible using direct numerical simulation, it is not without its shortcomings. During the ADI phase of the solver we routinely need to perform solves over the slower varying dimensions of the grid space and hence incur a severe cache miss and vectorization penalty. In this thesis we attempt to address some of these problems, including data layout transformations along with code transformations that can ease the problem. In the following section we will formally define the problem.
Another trend used in many scientific kernels is using a combination of coarse-grained parallelism along with fine-grained parallelism. The 3D matrix distribution to multiple processors introduces coarse-grained parallelism in the computational task. But the current implementation lacks effective use of fine-grained parallelism or any support to exploit such a scheme. Previously, many scientific kernels attempted to introduce some degree of fine-grained parallelism by introducing a hybrid of MPI+OpenMP approach in the kernels. Thus when the processes are spawned, there would be some master processed on each node of the supercomputing systems and these masters would spawn off multiple threads that would run over their node in the Symmetric Multi Processors (SMP) configuration. Today with the stagnation of per core clock rates, to achieve good speedup we need to exploit multiple levels of parallelism, and seek to utilize many core processors, or accelerators [3]. Traditional CPUs have a small number of cores that are extremely complex, fast and are capable of doing some sophisticated scheduling and cache manipulation. In contrast, accelerators offer a large number of low frequency execution units with extremely high degrees of parallelism. In this thesis, we also examine a particularly computationally intensive kernel of the FDL3DI code and evaluate the use of the new Intel Xeon Phi accelerator for this kernel.
Chapter 2: Problem Statement

2.1 Data Layout Transformations

2.1.1 ADI Scheme

For the entire 3 dimensional grid that each processor is responsible for, the computation is done in an alternate direction implicit fashion. Thus, for each processor, the input 3D matrix is solved by partitioning it to 3 types of separate 2D grids. Thus we have the following grid layout:

1. Grid along I
   a. #Grids = 1..IMAX
   b. Each grid ranges from
      i. J= 1..JMAX and
      ii. K= 1..KMAX
   c. Each grid can be visualized as a (*, J, K) 2D grid slice.

2. Grid along J
   a. #Grids = 1..JMAX
   b. Each grid ranges from
      i. I= 1..IMAX and
      ii. K= 1..KMAX
   c. Each grid can be visualized as a (I, *, K) 2D grid slice.
3. Grid along K
   a. #Grids = 1..KMAX
   b. Each grid ranges from
      i. I= 1..IMAX and
      ii. J= 1..JMAX
   c. Each grid can be visualized as a (I, J, *) 2D grid slice.

   We must emphasize that this is just a logical division that is carried out in terms of the way the code accesses the 3D grid data structure. The data on the grid is still stored as a set of 3 dimensional matrices of double precision real numbers.

2.1.2 Performance bottlenecks with the ADI scheme

   As we have described in section 2.1 above, the ADI scheme logically divides up the 3D grid into the three types of 2D grids for solving the equations. Now, since the data is still organized in the 3D fashion, at least one of the 2D grids will have access to data elements in a non-unit stride fashion. Which of the three types of grids have non-unit stride access is determined by the way arrays are linearized in the implementation language.

   1. Fortran
      o Linearization: Column Major
      o Hence I will be the fastest varying dimension in the 3D Matrix (I, J, K)
      o Slices along I, have J and K as the “iteration” dimensions and hence will have to access elements in a non unit stride fashion.
2. C
   
   - Linearization: Row Major
   - K will be the fastest varying dimension in the 3D Matrix (I, J, K)
   - Slices along K, have I and J as the varying dimensions and hence will have to access elements in a non unit stride fashion.

   Since the implementation that we deal with is in Fortran we are primarily interested in the slices along I, which exhibit non-unit stride accesses. We note that in case of the slices along J and K, non-unit stride accesses could occur if the inner-loops of computations access elements varying along some dimension other than I, but loop permutations can be used in such cases to arrange all the computation in innermost loops to be done along the fastest varying dimension I.

2.1.3 Problems with non unit stride access

   Performance of code having non-unit stride access is usually extremely low due to the following reasons

   1. Inability to vectorize code

   Code having non-unit stride cannot be vectorized or can be vectorized only partially. Table 1 gives the vector lengths for today’s popular instructions for the Intel x86 family of processor architectures. We can see that for double precision floating point operations, as in our implementation, we will have a performance penalty of up to 2x for SSE4, 4x for AVX and up to 8x for AVX-512 (used on Intel MIC) simply because we are unable to vectorize code due to non unit stride access.
2. **Cache performance**

Non-unit stride access also results in a cache access penalty. In most scientific kernels we deal with considerably large inputs matrices. Thus if we access a 3 dimensional matrix $A(I, J, K)$ along the non unit stride dimension of $J$, the two varying elements $J$ and $J +/- 1$ might not be in the same cache line. For many scientific kernels that have access patterns like stencil computation, the access is usually $(+/-) c$, where $c$ is some constant, over the iteration space. This means that if we are iterating over non-unit stride dimensions, there is absolutely no re-use of data belonging to same cache lines. Thus we can see a larger number of costly cache misses. This can rise up to a factor of $(Cache \ line \ size/ \ size \ of \ 1 \ element \ of \ data)$ when compared to unit stride access.

2.2 Fine grained parallelism and use of Xeon Phi

2.2.1 Potential for fine grained parallelism

The implementation breaks up the 3 dimensional data space into multiple smaller 3 dimensional data spaces, where each data space is computed independent of
others and the boundary solution is periodically exchanged in order to propagate the effect of computation.

This uses very coarse-grained parallelism. The smaller data grids distributed to each processor itself can range from (120*120*120) 3 dimensional grids. Thus we can see that there could be scope for introducing fine grained parallelism within this computation without having adverse effect on the coarse grained parallelism already exploited.

2.2.2 Xeon Phi

Intel MIC (Many Integrated Cores) architecture combines many Intel CPU cores and is aimed at achieving high throughput performance for highly parallel, High Performance Computing (HPC) workloads [4]. Based on the whitepaper [4], a key attribute of the microarchitecture is that it is built to provide a general-purpose programming environment similar to the Intel® Xeon® processor programming environment. The Intel Xeon Phi co-processors based on the Intel MIC architecture run a full service Linux operating system, support x86 memory order model and IEEE 754 floating-point arithmetic, and are capable of running applications written in industry-standard programming languages such as Fortran, C, and C++. A rich development environment that includes compilers, numerous libraries such as threading libraries and high performance math libraries, performance characterizing and tuning tools, and debuggers supports the coprocessor. According to [5], the coprocessor features more than 50 cores clocked at 1 GHz or more, supporting 64-bit x86 instructions. These in-order cores support four-way hyper-threading, resulting in
more than 200 logical cores. Cores of Intel Xeon Phi coprocessors are interconnected by a high-speed bidirectional ring, which unites L2 caches of the cores into a large coherent aggregate cache over 25 MB in size. The coprocessor also has over 6 GB of onboard GDDR5 memory. The speed and energy efficiency of Intel Xeon Phi coprocessors comes from its vector units. Each core contains a vector arithmetic unit with 512-bit SIMD vectors supporting a new instruction set called Intel Initial Many-Core Instructions (Intel IMCI). The Intel IMCI includes, among other instructions, the fused multiply-add, reciprocal, square root, power and exponent operations, commonly used in physical modeling and statistical analysis. The theoretical peak performance of an Intel Xeon Phi coprocessor is 1 TFLOP/s in double precision. This performance is achieved at the same power consumption as in two Intel Xeon processors, which yield up to 300 GFLOP/s. Due to the wide vector units, and abundance of cores, we can have a good amount of fine grained parallelism that can be exploited from the Xeon Phi hardware for our application. Over the next few sections we will see the types of programming models available on the Xeon Phi, the models explored as a part of this work and evaluation of performance for the particular kernel.

2.2.3 Exploiting parallelism with Xeon Phi

As demonstrated by [6], the amount of parallelism offered by Xeon Phi can be so large that the length of most OpenMP loops of standard benchmarks are too short for balanced execution on the Xeon Phi. Moreover, as stated in section 2.2.2 The Xeon Phi has a frequency of only 1.1GHz as compared to ~2.4 to 3 GHz of most
standard processor cores today. But the Xeon Phi compensates for this by having many more cores, having 4 hyper-threads per core and by having extremely large vector length (see Table 1: Vector Instruction Sets- Vector Lengths). Thus in order to achieve maximum performance possible we are interested in the following aspects of Xeon Phi:

1. Large number of cores + hyper threading per core => Maximum 224 number of OpenMP threads concurrently executing code.

2. Large Vector lengths => Up to 8 double elements in single vector instruction.

At the same time we identify that performance can be hampered if we are not able to extract the full potential of 224 threads and if we cannot exploit vectorization of the code. In the next few sections we will see how we are able to vectorize the micro-kernel for Xeon Phi as well as develop an optimized version of the kernel with hybrid MPI+OpenMP.
Chapter 3: Design

3.1 Data Layout Transformations

3.1.1 Program Structure

As described in section 2 for this particular operation we focus our attention on SWEEPI which operates on slices of data having the same I value and iterates along the (J,K) plane for this data. The program structure is as follows:

DO I= ISTART, IEND
  • Setup of data
  • DO J=JSTART, JEND
    DO K=KSTART, KEND
    COMPUTE
    END DO
  ENDDO
  • Compute Function calls.
  • DO J=JSTART, JEND
    DO K=KSTART, KEND
    COMPUTE
    END DO
  ENDDO
  • Final Store results
The data structures used are as follows:

1. Data structures with only read accesses
   - All (I, J, K) matrices: ZETAX, ZETAY, ZETAZ, XIAC, XMU, EDDY, U, V, W, RHO, P, ETAX, ETAY, ETAZ, BFIELD, SIGMAXYZ

2. Data structures used as temporary storages of intermediate results
   - TEMP (J, K, CONST)
   - WRK (J, K, CONST, CONST)

3. Data structures with writes that need to be preserved
   - RHS (I, J, K)

3.1.2 Data Reorganization

For the data structure and program structure in section 3.1.1 above, we have a good amount of re-use of data for the same access. Thus the approach we pursue is to have one non-unit stride access to data and cache it in a temporary unit stride data structure. Thus we develop multiple mirroring data structures that cache all the data of the original data structure for the current I slice. Moreover, we do not need to keep these data structures persistent over multiple slices of I plane. This means the same data structure can be reused for all the data layout transformations over multiple iterations of I. We expect the performance to be better since we will end up having lower cache misses and good vectorization. Thus for the data structures with read-only access we have the following mapping
ETAX(I, J, K) \rightarrow TEMPETAX(K, J)

Now, observe that when we are reading ETAX(I, J, K) we will already have ETAX(I+1, J, K) available in cache in the same line. For now we will only do the data layout transformation for the I plane and discard I+1 plane, which is already present in cache. This means that during the next I+1 iteration we could end up having another cache miss for the same cache line.

3.1.3 Loop Permutation and unrolling

As described in section 3.1.2, we perform a data layout transformation on the I plane for every I iteration. But this results in multiple cache misses when we are re-organizing data for successive I planes. The alternative approach we use is that of loop permutation and unrolling. Thus we permute the fastest varying I dimension to be the innermost loop and unroll this loop. Thus the earlier structure of code is

1. DO I= ISTART, IEND
   DO J= JSTART, JEND
   DO K=KSTART,KEND
   COMPUTE

is now transformed to

1. Data Layout transformations
2. DO I= ISTART, IEND, UNROLL-LENGHT
   DO J= JSTART, JEND
   DO K=KSTART,KEND
   DO I2= I, I+UNROLL-LENGHT
   COMPUTE
3. Data Layout transformations

In the optimum unroll factor we will utilize all elements belonging to same cache line in a single miss on that cache line. But as we observe some of the temporary data structures are not particularly permutation and unroll friendly. Thus we will have to do a data layout transformation to expand these data structures from \((J, K)\) to \((I_{UNROLL}, J, K)\). But as we can see the cost of this transformation is lower than that for the data layout transformation suggested in the earlier section.

3.1.4 Implementation issues

For the data layout transformation, we need to create additional 2 dimensional matrices for all the data structures that need to be transformed. This means there are additional memory requirements. This could be a problem if the problem size exhausts the memory available on the system. In our tests we do not face this issue.

For the loop permutation and unrolling we face significant issues in the data transformation required for the output data structures. The iteration space is regular but unknown at compile time and the number of iterations is equal to the problem size. Thus we need to perform some peeling of the loop to execute the remaining iterations after the unrolling is done. The original loop body is more than 700 lines of Fortran code. Thus repeating the loop body could result in code explosion and significant maintenance issues in the future. Alternatively, the following approach is taken. After the permuted and unrolled loop body execution we check if we have any iterations left, do the necessary setup modification to UNROLL-LENGTH and jump inside the original loop body. Along with this we also have significant challenges with the data
layout transformations since there are multiple routines accessing these data structures called from inside the core computation being transformed. This means that we will have to transform the procedures themselves and propagate the unrolling in these routines as well. We identify total of 15 routines that were required to transform and the total codebase after the code transformation was around 3000 lines. Moreover since these routines are used in their original implementation, we end up with two implementations of all unrolled routines present in the code.

3.2 Optimizing on Xeon Phi

3.2.1 Vectorization

As seen in section 2.1.3, vectorization is extremely important and can result in quite a lot of performance gain based on architecture used. Especially for the Xeon Phi we observe that with its large vector lengths, efficient vectorization of code is extremely important to achieve any good performance. But it might not always be possible. The following conditions are recommended for vectorization of code:

1. Unit stride to data structure in consecutive iteration.

2. No loop carried dependencies.

Here loop carried dependencies is where the same data is accessed in two or more iterations and at-least of one of these accesses is a write operation on the data. If such patterns exist then clearly we need to wait for the first iteration accessing the data to execute before we can start execution of the other iterations accessing the data. It
might be possible to resolve strided access by data transformations or by using loop permutations but usually loop carried dependencies cannot be solved by compilers and need changes in the broad scope of how things are computed.

For this part we focus our attention on 3 variants of similar routines, XIROEFLUX, ETROEFLUX, ZTROEFLUX. Based on the vectorization report of the three routines using Intel’s compiler for Fortran “ifort”, the compiler is not able to vectorize these codes due to existence of dependences. We do exhaustive dependence analysis of the source code to identify the dependence and find the following access patterns among the routine XIROEFLUX:

For each iteration along I

1. Read from 11 x 3 dimensional arrays and 1 x 3 dimensional output array into scalars.
2. Computation on scalars
3. Write into the same 3 dimensional output array.

Notice that there is a read/write access on a single 3 dimensional array. But we observe that the read/write do not occur on the same elements of the same array. The array is read at the indices (I, 3, *) and (I, 4, *) and written only into (I, 5 *). Thus although we do access the same array, we are never accessing the same element for both read and write. Thus really there is no true data dependence that is present. The access pattern can be visualized in Figure 5. The access patterns for the other routines, ETROEFLUX and ZTROEFLUX, are similar. We are able to vectorize the code by
using the “!DIR$ SIMD” compiler directive that instructs the compiler to vectorize the loop without any dependence analysis or performance heuristics.

![Data structure with both read & write accesses](image)

*Figure 5: Data structure with both read & write accesses*

### 3.2.2 SMP parallelism with OpenMP

Based on the structure defined in the earlier section, we can see that along with vectorization we can also go for SMP parallelism since there are no loop carried dependences. We do this by the work-sharing construct on the single DO loop present in the code. Thus for possible “T” threads and for the iteration space I= ISTART to IEND, we will distribute the iterations onto T threads.

One of the main performance bottlenecks possible with OpenMP is the possibility of False Sharing [7]. False sharing occurs when n threads are reading from
and/or writing to the same cache line. This results in the cache line getting updated and invalidated repeatedly, which results in cache trashing. This can cause significant degradation in performance to the extent that the performance may deteriorate to the level of serial application or could possibly be even worse than the serial application. False sharing is shown in Figure 6.

![Figure 6: False Sharing](image)

In order to ensure no false sharing takes place we need to ensure that no two threads have read/write operations to the same cache line. Observing the access patterns of the data structure used we determine that setting the OMP_SCHEDULE to static will divide up the iteration space such that chunks of consecutive iterations will be allocated to the same threads. Thus two threads will defer in the iteration they execute with the largest possible difference. Thus different threads will access data belonging to
different cache lines. Thus we will have almost no false sharing except for the elements on the boundary of each thread’s iteration space. The division is as shown in Figure 7.

![Figure 7: OpenMP Static Schedule allocation](image)

### 3.2.3 Offload from host to MIC

In many scientific applications as in our kernel, we are interested in offloading only the chunk of the computation to the accelerators. This is because of the nature of accelerators, giving high performance for fine-grained parallel vector code but not being particularly efficient for non-vector serial code. The clock frequency for a single core is 1.1GHz. Thus a serial code without vectorization is expected to be at least ~3 times slower on the MIC as opposed to normal CPUs. On the other hand the peak performance of MIC, for double precision numbers is 1 TeraFlops as opposed to around 300 GigaFlops on today’s best server processors. Hence the model is one where we have the compute intensive, fine-grained parallel tasks offloaded to MIC and other normal workloads executing on host. For offloading compute to MIC, the main operations are as follows:

1. Communicate data to Xeon Phi
2. Compute
3. Communicate results back to CPU

Since all of the communication occurs over PCI-e bus we expect extremely communication as compared to the compute. This is critical because it implies that only procedures having extremely high compute intensity per memory operation can be effectively offloaded to MIC. The underlying idea is that the time required communicating input to and output from the MIC should be much less than the time saved in computing on the MIC. Only then will we be able to use MIC effectively in offload mode even while paying the communication costs. In order to be as efficient as possible MIC allows the programmer to specify the data structures that we need to communicate to and from the MIC. Based on our analysis of the original code we specify “IN” and “OUT” communication to and from MIC using OpenMP directives. The communication cost can be expressed as

1. CPU to MIC communication
   - 11 * 3 Dimensional input arrays
   - 2 * 5 * 3 Dimensional array used for in/out

2. MIC to CPU
   - 5 * 3 Dimensional array used for in/out

As described in earlier section one of the 3 dimensional array is used for input/output but we use different levels of the 3 dimensional array for input and different levels for output. We can visualize the use of the data structure as in Figure 5.
3.2.4 Challenges in implementation for Xeon Phi

We face significant challenges in programming for the MIC. In theory, due to its x86 origins, programming for mic is as simple as setting the “-mmic” compiler flag while compiling your source code. In practice though, although this compiler level porting that can be done is not at all efficient and will result in extremely low performance code. This is because the compiler is bound to be extremely conservative in any compiler level optimizations and ends up producing the most naïvely optimized code. Further setting proper environment variables is crucial on the MIC. This issue is further exacerbated by the extreme number of threads on the MIC and the range of performance possible by various thread configurations for the same application [6]. The environment variables play a crucial load in the offload code as well. We observe that setting the “MIC_USE_2MB_BUFFERS” to a reasonable value of 16KB decreased the offload communication time dramatically for larger communications. The performance reported in later sections should be considered to be the “best setup” performance in all cases. Although we have optimized the setup and environment for our application, we do not evaluate the effect of these on the execution on Phi, since the optimizations performed are specific to our application and a generic evaluation of these is not the scope of this work.
Chapter 4: Experimental Results

4.1 Experimental Setup

We run experiments on the Ohio Supercomputing Center [8] Oakley [9] cluster as well as the internal cluster machine, Hydra, at the High Performance Computing Research Lab at the Ohio State University (OSU). The configurations are as follows:

4.1.1 Oakley

Oakley contains a total of 8,328 cores with 12 cores per node and 48 GigaBytes of memory per node [9]. Each node has the 2.66 GHz Intel Xeon x5650 CPU’s. It has a QDR IB interconnect with low latency, high throughput and high quality of service [9]. All the nodes are homogenous. Any reference in the material to “Oakley” or “reference system” or “reference performance” refers to use of this system.

4.1.2 Hydra

Hydra is an internal cluster node at the High Performance Computing Research Lab at OSU. The system is equipped with Xeon Phi Coprocessor 3120P card, which acts as an accelerator. The host is an Intel Xeon CPU E5-2650 which has 4 cores with 2 hyper threads per core. Any reference to systems “MIC”, “hydra” or “Phi” refers to this system.
4.2 Data Layout Transformations

For this set of experiments we run the optimizations on a grid size of 643x395x75. All measurements for this section are carried out on the Oakley system.

4.2.1 Data Reorganization

We get the following performance for the entire application:

<table>
<thead>
<tr>
<th># Processes</th>
<th>Original (sec)</th>
<th>Data Reorg (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11985.40</td>
<td>11655.99</td>
<td>1.03</td>
</tr>
<tr>
<td>4</td>
<td>8470.89</td>
<td>8290.78</td>
<td>1.02</td>
</tr>
<tr>
<td>8</td>
<td>5854.17</td>
<td>5689.67</td>
<td>1.03</td>
</tr>
<tr>
<td>12</td>
<td>3995.62</td>
<td>3898.31</td>
<td>1.02</td>
</tr>
<tr>
<td>24</td>
<td>1876.53</td>
<td>1834.59</td>
<td>1.02</td>
</tr>
<tr>
<td>48</td>
<td>981.52</td>
<td>966.93</td>
<td>1.02</td>
</tr>
<tr>
<td>96</td>
<td>477.15</td>
<td>472.18</td>
<td>1.01</td>
</tr>
<tr>
<td>120</td>
<td>299.03</td>
<td>299.31</td>
<td>1.00</td>
</tr>
<tr>
<td>320</td>
<td>152.63</td>
<td>150.71</td>
<td>1.01</td>
</tr>
</tbody>
</table>

*Table 2: Data reorganization - Application Execution Time & Speedup*

We see that although we have some speedup, we do not achieve very favorable results. This is due to the cache usage factors described in the design section. Since this is the overall performance of the whole application the gain calculated is further decreased since our optimizations are only for a sub part of the whole kernel. For the kernel optimized, the data is as follows:
<table>
<thead>
<tr>
<th>#Processes</th>
<th>Cycles(I)</th>
<th>Cycles(E)</th>
<th>Speedup(I)</th>
<th>Speedup(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.69E+13</td>
<td>1.18E+13</td>
<td>1.07</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>2.41E+13</td>
<td>1.72E+13</td>
<td>1.09</td>
<td>1.11</td>
</tr>
<tr>
<td>8</td>
<td>3.13E+13</td>
<td>2.30E+13</td>
<td>1.11</td>
<td>1.14</td>
</tr>
<tr>
<td>12</td>
<td>3.61E+13</td>
<td>2.70E+13</td>
<td>1.09</td>
<td>1.11</td>
</tr>
<tr>
<td>24</td>
<td>3.48E+13</td>
<td>2.60E+13</td>
<td>1.07</td>
<td>1.09</td>
</tr>
<tr>
<td>48</td>
<td>3.51E+13</td>
<td>2.63E+13</td>
<td>1.04</td>
<td>1.04</td>
</tr>
<tr>
<td>96</td>
<td>3.33E+13</td>
<td>2.47E+13</td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td>120</td>
<td>3.00E+13</td>
<td>2.33E+13</td>
<td>1.02</td>
<td>1.02</td>
</tr>
<tr>
<td>320</td>
<td>1.88E+13</td>
<td>1.24E+13</td>
<td>1.07</td>
<td>1.15</td>
</tr>
</tbody>
</table>

*Table 3: Data reorganization - Kernel Performance & Speedup*

In above table (I) indicates Inclusive profiling of all subroutines and (E) indicates Exclusive profiling of SWEEPI routine of the kernel. As we can see we get a speedup of 1.15x on the subroutine and 1.03x on the entire kernel.

### 4.2.2 Loop Permutation with Unrolling

We perform analysis with unrolling factor of 4 to 32, doubling the unroll factor each time.

1. **Unroll factor 4**

   As we see in Table 4, we get a significant slowdown in performance for this case. Motivation to permute and unroll the loops is mainly to reduce the cache misses. But we observe that for an unroll factor of 4 we incur greater cache misses due to the layout transformations required and thus end up with worse performance.
We observe that the performance is better than unrolling by 4 but still cannot compensate for the additional cost of data transformations associated with
unrolling. As we increase the unrolling factor our cache line utilization is expected to increase and performance is expected to be better.

3. Unroll factor 16

<table>
<thead>
<tr>
<th>#Processes</th>
<th>Original (sec)</th>
<th>Unroll 16 (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11985.40</td>
<td>10938.08</td>
<td>1.10</td>
</tr>
<tr>
<td>4</td>
<td>8470.89</td>
<td>7813.24</td>
<td>1.08</td>
</tr>
<tr>
<td>8</td>
<td>5854.17</td>
<td>5278.77</td>
<td>1.11</td>
</tr>
<tr>
<td>12</td>
<td>3995.62</td>
<td>3607.91</td>
<td>1.11</td>
</tr>
<tr>
<td>24</td>
<td>1876.53</td>
<td>1719.55</td>
<td>1.09</td>
</tr>
<tr>
<td>48</td>
<td>981.52</td>
<td>901.37</td>
<td>1.09</td>
</tr>
<tr>
<td>96</td>
<td>477.15</td>
<td>455.29</td>
<td>1.05</td>
</tr>
<tr>
<td>120</td>
<td>299.03</td>
<td>288.51</td>
<td>1.04</td>
</tr>
<tr>
<td>320</td>
<td>152.63</td>
<td>167.47</td>
<td>0.91</td>
</tr>
</tbody>
</table>

Table 6: Unroll factor 16- Application Execution Time

<table>
<thead>
<tr>
<th>#Processes</th>
<th>Cycles(I)</th>
<th>Cycles(E)</th>
<th>Speedup(I)</th>
<th>Speedup(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.28E+13</td>
<td>6.55E+12</td>
<td>1.42</td>
<td>1.95</td>
</tr>
<tr>
<td>4</td>
<td>1.85E+13</td>
<td>9.10E+12</td>
<td>1.41</td>
<td>2.11</td>
</tr>
<tr>
<td>8</td>
<td>2.35E+13</td>
<td>1.16E+13</td>
<td>1.48</td>
<td>2.26</td>
</tr>
<tr>
<td>12</td>
<td>2.65E+13</td>
<td>1.40E+13</td>
<td>1.48</td>
<td>2.14</td>
</tr>
<tr>
<td>24</td>
<td>2.68E+13</td>
<td>1.43E+13</td>
<td>1.39</td>
<td>1.98</td>
</tr>
<tr>
<td>48</td>
<td>2.61E+13</td>
<td>1.43E+13</td>
<td>1.39</td>
<td>1.92</td>
</tr>
<tr>
<td>96</td>
<td>2.76E+13</td>
<td>1.56E+13</td>
<td>1.24</td>
<td>1.64</td>
</tr>
<tr>
<td>120</td>
<td>2.63E+13</td>
<td>1.43E+13</td>
<td>1.16</td>
<td>1.65</td>
</tr>
<tr>
<td>320</td>
<td>2.43E+13</td>
<td>1.29E+13</td>
<td>0.83</td>
<td>1.10</td>
</tr>
</tbody>
</table>

Table 7: Unroll factor 16- Kernel Performance Data

As expected, as we increase the unroll factor to 16, we get improved performance and achieve speedup of \textbf{1.10x} over the base version. Again this
speedup is over the entire application. We get a better picture of the effectiveness of our optimizations by looking at the performance of the kernel. As we see in Table 7: Unroll factor 16- Kernel we get up to 1.48x speedup on the sub kernel.

4. Unroll factor 32

<table>
<thead>
<tr>
<th>#Processes</th>
<th>Original (sec)</th>
<th>Unroll 32 (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>11985.40</td>
<td>10341.88</td>
<td>1.16</td>
</tr>
<tr>
<td>4</td>
<td>8470.89</td>
<td>7211.42</td>
<td>1.17</td>
</tr>
<tr>
<td>8</td>
<td>5854.17</td>
<td>4805.57</td>
<td>1.22</td>
</tr>
<tr>
<td>12</td>
<td>3995.62</td>
<td>3291.14</td>
<td>1.21</td>
</tr>
<tr>
<td>24</td>
<td>1876.53</td>
<td>1566.99</td>
<td>1.20</td>
</tr>
<tr>
<td>48</td>
<td>981.52</td>
<td>826.03</td>
<td>1.19</td>
</tr>
<tr>
<td>96</td>
<td>477.15</td>
<td>427.08</td>
<td>1.12</td>
</tr>
<tr>
<td>120</td>
<td>299.03</td>
<td>264.30</td>
<td>1.13</td>
</tr>
<tr>
<td>320</td>
<td>152.63</td>
<td>151.78</td>
<td>1.01</td>
</tr>
</tbody>
</table>

*Table 8: Unroll factor 32- Application Execution Time*

<table>
<thead>
<tr>
<th># Processors</th>
<th>Cycles(I)</th>
<th>Cycles(E)</th>
<th>Speedup(I)</th>
<th>Speedup(E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4.57E+12</td>
<td>2.33E+12</td>
<td>3.98</td>
<td>5.47</td>
</tr>
<tr>
<td>4</td>
<td>5.91E+12</td>
<td>3.10E+12</td>
<td>4.43</td>
<td>6.17</td>
</tr>
<tr>
<td>8</td>
<td>1.46E+13</td>
<td>7.74E+12</td>
<td>2.38</td>
<td>3.39</td>
</tr>
<tr>
<td>12</td>
<td>1.58E+13</td>
<td>9.23E+12</td>
<td>2.48</td>
<td>3.25</td>
</tr>
<tr>
<td>24</td>
<td>1.65E+13</td>
<td>9.61E+12</td>
<td>2.25</td>
<td>2.94</td>
</tr>
<tr>
<td>48</td>
<td>1.65E+13</td>
<td>9.74E+12</td>
<td>2.21</td>
<td>2.82</td>
</tr>
<tr>
<td>96</td>
<td>1.93E+13</td>
<td>1.20E+13</td>
<td>1.78</td>
<td>2.14</td>
</tr>
<tr>
<td>120</td>
<td>1.69E+13</td>
<td>9.82E+12</td>
<td>1.81</td>
<td>2.40</td>
</tr>
<tr>
<td>320</td>
<td>1.60E+13</td>
<td>9.00E+12</td>
<td>1.26</td>
<td>1.57</td>
</tr>
</tbody>
</table>

*Table 9: Unroll factor 32- Kernel Performance Data*
As expected we see better performance with a maximum speedup of 1.21x on the entire application. The data for the specific subkernel optimized is as follows with a maximum speedup of 4.43x.

### 4.3 Optimizing on Xeon Phi

#### 4.3.1 Vectorization

We evaluate performance of the no vector code on Oakley (reference) vs vector code on Oakley vs vector code on Xeon Phi. We have a single I loop over which all computation is being performed. The “Problem Size” after this refers to the iteration space for this I loop. We run test on problem sizes ranging from 1024 to 10240000. The execution times are as follows.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Oakley reference (sec)</th>
<th>Oakley vector (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>1.00E-03</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>10241</td>
<td>5.00E-03</td>
<td>3.00E-03</td>
<td>1.67</td>
</tr>
<tr>
<td>102401</td>
<td>4.80E-02</td>
<td>2.80E-02</td>
<td>1.71</td>
</tr>
<tr>
<td>1024001</td>
<td>0.25</td>
<td>0.14</td>
<td>1.75</td>
</tr>
<tr>
<td>10240001</td>
<td>2.68</td>
<td>1.54</td>
<td>1.74</td>
</tr>
</tbody>
</table>

*Table 10: Vectorization Execution Time - Oakley*

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Xeon Phi no vector (sec)</th>
<th>Xeon Phi vector (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>10241</td>
<td>3.00E-02</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>102401</td>
<td>0.5</td>
<td>7.00E-02</td>
<td>7.14</td>
</tr>
<tr>
<td>1024001</td>
<td>7.18</td>
<td>1.01</td>
<td>7.11</td>
</tr>
<tr>
<td>10240001</td>
<td>71.65</td>
<td>10.17</td>
<td>7.05</td>
</tr>
</tbody>
</table>

*Table 11: Vectorization Execution Time - Xeon Phi*
On Oakley, with SSE we get a speedup of 1.75x with a vector length of just 2 doubles. The results on Xeon Phi are even better with the vector lengths of 8 and we get a maximum speedup of 7.14x.

4.3.2 OpenMP

As described in the above sections we have up to 56 cores available on the Xeon Phi at the hydra node. Moreover each of these 56 cores can support up to 4 hyper threads. We perform an exhaustive analysis of the performance of OpenMP parallelism on the vectorized code on Xeon Phi with various thread configurations in order to find the optimum configuration for various problem sizes.

For convenience, we present only the best performance achieved for each problem size along with graphs showing the performance on various core and thread configurations. OpenMP schedule used is STATIC.

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Serial Vector (sec)</th>
<th>OpenMP + Vector (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1025</td>
<td>---</td>
<td>3.22E-02</td>
<td>----</td>
</tr>
<tr>
<td>10241</td>
<td>---</td>
<td>2.55E-02</td>
<td>----</td>
</tr>
<tr>
<td>102401</td>
<td>7.00E-02</td>
<td>4.43E-02</td>
<td>1.58</td>
</tr>
<tr>
<td>1024001</td>
<td>1.01</td>
<td>1.34E-01</td>
<td>7.53</td>
</tr>
<tr>
<td>10240001</td>
<td>10.17</td>
<td>3.10E-01</td>
<td>32.78</td>
</tr>
</tbody>
</table>

*Table 12: OpenMP Execution Time- Xeon Phi*

As we can see the utilization on MIC increases as we increase our problem size and we get the maximum speedup of 32.78x using OpenMP. Moreover,
observe that as we increase the problem size from 1M to 10M we get only 2x slowdown rather than the expected 10x. This is because the larger problem size is able to efficiently utilize more cores on the Xeon Phi. For 1M problem we get peak performance at 32 cores while the 10M problem is able to utilize the 54 cores on the Xeon Phi with 1 thread per core.

It is interesting to observe the effect core and hyperthreading utilization as we vary the problem size of execution on the Xeon Phi. Since each test contains 240 data points it is impractical to represent the tabled data points here. Instead we present plot graphs of performance on various thread and core configurations for a range of problem sizes.

1. Problem Size=102400

As we see the performance keeps decreasing as we spawn more threads and try to utilize more cores and hyper threading of Xeon Phi. This is because of the extremely problem size. We observe similar trends for smaller problem sizes. Due to the small problem size, the cost of spawning threads on Xeon Phi proves to be greater than the speedup due to multiple threads. Hence the best performance we get is at 7 cores with 1 thread per core. Increasing the threads or cores beyond this point just decreases the performance.
2. Problem Size= 1024000

We see that as we increase the problem size, we start getting better core and thread utilization. For this problem size the best performance is obtained at 12 cores with 2 hyper threads per core. Moreover, we observe that as we increase the cores and threads per core (tpc), initially we get better performance but after we hit the peak point increasing both the values just reduces the performance. Thus the problem is large enough to give us better utilization but still not large enough to fully utilize all the cores of Xeon Phi.

*Figure 8: Performance variation on MIC- 102400 Problem Size*
3. Problem Size= 10240000

With further increase in problem size we see really good core and thread utilization of MIC. Best performance is achieved at 54 cores and 1 thread, just shy of the full 56 cores of MIC. Performance peaks at 41 cores for 2tpc, 31 cores for 2tpc, 25 cores for 4tpc. Moreover, performance at these peaks is extremely close to the aggregate peak at 54 cores, 1 thread. This gives us an interesting insight into potential optimization of offloads. Notice that the performance for 25 cores with 4 tpc is extremely close to peak. This is important for offload onto MIC. Since we cannot have multiple offloads onto same core, we can still accommodate
multiple offloads mapped onto different cores and still achieve extremely good speedups. We can have weak scaling where two processes are offloading the problem onto 25 distinct cores and we still expect to get the same speedup. Similar techniques are extended for having up to 4 processes offloading to a specific sub-part of the accelerator in our offload experiments in the next subsection.

4.3.3 Offload from host to MIC

As described in earlier section with fine grained control on the cores to which we offload, we offload from 4 MPI processes running on host onto a single MIC and evaluate the performance against both OpenMP
and MPI implementations of best performance on 1 node (12 cores) running on Oakley. The performance is as follow:

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>MIC-Offload Time (sec)</th>
<th>Communication Time (sec)</th>
<th>MIC-Exec Time (sec)</th>
<th>Oakley Exec Time (sec)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4100</td>
<td>0.001</td>
<td>0.001</td>
<td>0.000</td>
<td>9.10E-05</td>
<td>0.04</td>
</tr>
<tr>
<td>40960</td>
<td>0.003</td>
<td>0.002</td>
<td>0.001</td>
<td>9.21E-04</td>
<td>0.24</td>
</tr>
<tr>
<td>409600</td>
<td>0.016</td>
<td>0.012</td>
<td>0.003</td>
<td>9.53E-03</td>
<td>0.58</td>
</tr>
<tr>
<td>4096000</td>
<td>0.133</td>
<td>0.102</td>
<td>0.030</td>
<td>6.06E-02</td>
<td>0.45</td>
</tr>
</tbody>
</table>

*Table 13: MIC Offload Performance*

We note that the communication performance increases by ~10x for our kernel if we allocate the matrices on Xeon Phi before the kernel execution rather than allocating the memory while we transfer the data for computing. The performance reported above is this improved performance with this optimization. As we can see from above table, we get a slowdown on offload to MIC. Moreover, we observe that we do get a pretty good increase in performance if we compare just the compute times on MIC vs Oakley. Thus, even with improved communication, we still have a communication bottleneck for offloading onto Xeon Phi. We can model the same data in the graph as below:
4.3.4 Improving offload performance

We compute the achieved bandwidth for our computation for offloading onto Xeon Phi:

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Bandwidth (GBps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4100</td>
<td>0.13</td>
</tr>
<tr>
<td>40960</td>
<td>0.82</td>
</tr>
<tr>
<td>409600</td>
<td>1.70</td>
</tr>
<tr>
<td>4096000</td>
<td>2.07</td>
</tr>
</tbody>
</table>

*Table 14: Offload Bandwidth*

If we compare this with the theoretical peak performance of PCI-e 2.0, which is 16GBps [10], we are barely using 1/8th of the total bandwidth. Also, the
observed peak performance on Xeon Phi is ~6GBps [11]. Thus, clearly, if we are able to reach this peak PCI-e performance then, we should get better performance on Xeon Phi for our kernel. Let us look at some of the speedups achieved on just the compute operations for a realistic modeling of the achievable speedups with reduced communication costs:

<table>
<thead>
<tr>
<th>Problem Size</th>
<th>Speedup compute</th>
</tr>
</thead>
<tbody>
<tr>
<td>4100</td>
<td>0.22</td>
</tr>
<tr>
<td>40960</td>
<td>0.85</td>
</tr>
<tr>
<td>409600</td>
<td>2.54</td>
</tr>
<tr>
<td>4096000</td>
<td>1.97</td>
</tr>
</tbody>
</table>

*Table 15: MIC Compute vs Oakley 1 Node speedup*

Thus we see that there is a scope for some modest speedup with reducing communication time by developing some sophisticated communication schemes and by the use of the future PCIe 3.0 technologies which double the peak communication bandwidth to 32 GBps [10].

Another trend in computer architectures is to integrate many core accelerators on the same die as CPU in order to optimize performance [12] [13] [14] [15]. Such architecture tries to avoid keeping two distinct memories for the host and accelerator and are becoming increasingly abundant. We already have such architectures already available today with the Intel Sandy Bridge processor and AMD APU architecture. Most of the current unified architectures still have
distinct CPU and GPU memories, which still have lower communication costs than over PCI-e bus. Current advances are towards building Heterogeneous Unified Memory Architectures (hUMA) with both the CPU and GPU residing on the same die and accessing the same memory [16]. In this type of single die architecture, we will incur minimal overhead to offload computation onto co-processor since the cost of data communication to and from the co-processor will be reduced and we will expect to get bandwidths greater than 40GBps available with todays L3 cache. Such architectures will be extremely advantageous for our kernel since in effect we will have this extra bandwidth for the data access and our current 2GBps over PCIe bus.
Chapter 5: Conclusion

This thesis has focused on optimizations to a compute-intensive kernel of the FDL3DI code, achieving performance gains of up to 1.22x. We have also obtained good insights into the effectiveness of data layout transformations. We have found that although there is a performance penalty associated with data layout transformations, there is potential for good speedup to be achieved, if coupled with other optimizations and transformations of code.

We have performed extensive analysis of a microkernel and ported it to the Xeon Phi, achieving significant speedup by the effective use of vectorization and fine-grained parallelism with OpenMP. Even though we achieve good speedup on the computation, we are unable to achieve a speedup on offload of the kernel due to the inability to achieve peak communication bandwidth over PCIe to the accelerator. With further advancements in communication techniques and computer architecture, these optimizations can be explored further and fully exploited.
Bibliography


[15] D. Lustig and M. Martonosi, "Reducing GPU offload latency via fine-grained CPU-