Automatic Transformations for Effective Parallel Execution on Intel Many Integrated Core

Kevin Stock
The Ohio State University
stockk@cse.ohio-state.edu

Louis-Noël Pouchet
The Ohio State University
pouchet@cse.ohio-state.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

Abstract

We demonstrate in this work the potential effectiveness of a source-to-source framework for automatically optimizing a sub-class of affine programs on the Intel Many Integrated Core Architecture. Data locality is achieved through complex and automated loop transformations within the polyhedral framework to enable parallel tiling, and the resulting tiles are processed by an aggressive automatic SIMD vector code generator.

We evaluate the effectiveness of this approach on tensor contraction kernels. We show a mean improvement of 1.86× over existing compiler techniques for single core performance, and combined with automatic parallelization we achieve 14.56× the performance of Intel's ICC Compiler on MIC.

1 Introduction

High performance software requires transforming code to exploit many aspects of modern hardware. In this work we attempt to address the increasing availability of many CPU cores, deep memory hierarchies, and increasingly wide SIMD vector instructions. The Intel Many Integrated Core (MIC) Architecture, evaluated here using Knights Ferry (KNF), is especially unique with regards to two of these aspects as it boasts 30 or more individual cores, and each is capable of operating on 512-bit wide SIMD vectors. As manually tuning every program to take full advantage of these resources is exceptionally tedious we propose a source-to-source compiler capable of high-level transformations to automatically improve data locality, parallelize across cores, and vectorize with aggressive optimizations to maximize their potential.

We focus on tensor contractions, a class of compute-intensive problems common in quantum chemistry. Tensor contractions are essentially generalized higher dimension matrix products where the tensors may be more than two dimensional and the contraction may be performed on multiple dimensions. We consider a subset of the tensor contraction kernels used in [11] which are extracted from CCSD [3], a common computation in *ab initio* electronic structure calculations. An example four dimensional tensor contraction could be written in C as:

```
for (i=0; i<N; i++)
 for (j=0; j<M; j++)
  for (k=0; k<O; k++)
   for (l=0; l<P; l++)
     C[l][j][k] += A[i][j][k]*B[i][l];
```

We refer to this kernel as `ljk-ijk-il` as a shorthand representation of the three arrays' accessor functions. In this case `i` is the dimension of contraction as it accesses both of the input tensors `A` and `B`. Vectorization occurs along the `k` dimension which accesses both `C` and `A` in unit-stride (i.e. the right-most array accessor in C/C++, or the left-most array accessor in Fortran/MATLAB). Because the `B` tensor is not accessed by the vectorized dimension, it is loaded as a scalar value and *splat* operation (i.e. broadcast) replicates it to all elements of a vector register. Also in this case, as `l` is the inner-most loop, loads of `A` can be hoisted out of the inner-loop and be reused across iterations as they will not change.

2 Synthesis of Code

To achieve high performance, a segment of code such as the example in Section 1 is given as input for high level transformations. The first transformation uses PTile [1] to extract tiles and enable parallelization, and then the tiles are vectorized using the vectorizer from [10] with a custom backend to generate variants of the code with KNF specific vector intrinsics.

2.1 PTile

PTile uses the polyhedral model to represent a loop nest as a system of affine inequalities, and through manipulation of those inequalities is able to create new code which maintains dependencies while extracting parameterized rectangular tiles and annotating the source code with OpenMP pragmas to enable parallelization. This enables the code to take advantage of the cache hierarchy and the many available cores. However, the cost of this code generation is more complicated loop structure and bounds which in turn makes it difficult for ICC to generate efficient SIMD vector code. Analysis of the assembly generated by ICC shows significantly less aggressive optimizations on the inner most loop when PTile is used: In the case of `ij-ik-kj` (i.e. matrix multiplication), fused multiply-adds are not used at all in the PTile code, where as the input code when compiled by ICC uses fused multiply-adds and more aggressive loop unrolling.

Since PTile enables tiles to take on any size, we developed a heuristic to select a tile size with auto-tuning across the space of possible tile sizes. Each dimension of the tile starts with a power of two (e.g. 32). As long as the data is not L1 cache resident, it halves the size of the dimension which produces the greatest reduction on data size. In the case of a tie, the longest dimension is selected. While this does not produce optimal tile sizes as seen in Section 3, it ensures that the data will fit in L1 cache and that all dimensions of the tile can be unroll-and-jammed, which is crucial for achieving good vector performance.

2.2 Vectorization

In the case of the tensor contractions considered for this work, different vectorized dimensions do not need to be considered as all the kernels have an obvious optimal dimension for vectorization where the output tensor and one of the input tensors are both accessed in unit-stride by the same dimension. Vectorization along this dimension for these contractions does not require in-register transpose (which enables vectorization along non-unit-stride dimensions) or reductions (for vectorization along contracted dimensions). The only instructions needed for these contractions are the vector intrinsics for unit-stride vector loads and stores, splats (i.e. a load replicating a scalar element to all elements of a vector), and fused multiply-adds. To optimally vectorize tiles generated by PTile we use a subset of the transformations considered in [10] applied to code vectorized along the optimal dimension. Specifically, we consider unroll-and-jam and permutations of the loop order.

Unroll-and-jam enables greater register reuse and thus a higher arithmetic intensity, but can produce a very large search space as all loops in tensor contractions are candidates for the transformation. To restrict the size of this space we only consider unrolling values which are divisors of their tiled dimension size, which additionally has the benefit of not generating code which requires expensive boundary case handling within the tile.

The effect of loop order on locality is minimal in the context of L1 cache resident tiles, however, the order is still vital to performance as it enables some loads and possibly stores to be hoisted out of the inner-most loop. In the case of tensor contractions all permutations of the loops are valid, but in the cases where the tensor contractions have more than 4 dimensions this can needlessly increase the search space as we are only interested in hoisting instructions out of the inner-most one or two loops. As such we only consider the permutations of the inner three loops for all choices of three loops from the entire loop nest to reduce the number of variants generated.

As mentioned, each of these transformations can potentially create a large search space, and as we consider the product of these two spaces there are many vectorized variants from which to select. To select the optimal variant we auto-tune across the space of variants.

In this work we do not support problem sizes where the vectorized dimension is not a multiple of the SIMD vector width to prevent unaligned memory accesses as KNF only supports aligned loads and stores. Use of masked loads and stores would enable dimensions to be any size but could convey a significant performance penalty as twice the loads and stores would be needed. Alternatively, tensors could be padded to a multiple of the vector width to work with the generated code.

As an example of this code generation, the tile code with intrinsics for the best performing variant of `ijk-il-jkl` is shown in Figure 1. The outer tiling code is omitted for clarity.

```
for (j = __loop_bound_10; j <= __loop_bound_11; j += 2) {
    for (l = __loop_bound_14; l <= __loop_bound_15; l += 16) {
        for (k = __loop_bound_12; k <= __loop_bound_13; k += 8) {
            B_temp[0]=_mm512_loadq(&B[j][k][l], 0, 0, 0);
            // 30 more loads of B omitted
            B_temp[31]=_mm512_loadq(&B[j+1][k][l+15], 0, 0, 0);

            for (i = __loop_bound_8; i <= __loop_bound_9; i += 1) {
                A_temp[0]=_mm512_loadq(&A[i][l], 0, 1, 0);
                // 14 more loads of A omitted
                A_temp[15]=_mm512_loadq(&A[i][l+15], 0, 1, 0);

                C_temp[0]=_mm512_loadq(&C[i][j][k], 0, 0, 0);
                C_temp[1]=_mm512_loadq(&C[i][j+1][k], 0, 0, 0);

                C_temp[0]= _mm512_madd231_pd(C_temp[0], A_temp[0], B_temp[0]);
                // 29 more madds omitted
                C_temp[1]= _mm512_madd231_pd(C_temp[1], A_temp[15], B_temp[31]);

                _mm512_storeq(&C[i][j][k], C_temp[0], 0, 0, 0);
                _mm512_storeq(&C[i][j+1][k], C_temp[1], 0, 0, 0);
            }
        }
    }
}
```

Figure 1: Example tile code generated with intrinsics.

3 Performance Evaluation

The experiments were performed on an Knights Ferry Intel Many Integrated Core Architecture with 30 cores clocked at 1.05 GHz. The kernels were compiled to natively target MIC using

ICC[1] (`icc -mmic -O3`). Kernels tested with OpenMP (`icc -mmic -O3 -openmp`) were set to use 30 threads. Auto-parallelization (`icc -mmic -O3 -parallel`) was tested, but even when loops were successfully parallelized the performance did not differ from the single core version. All of the kernels use `doubles` and the tensors were sized to occupy $\sim 6MB$ and require $\sim 125M$ operations for the standard dataset (Table 1), and for the large dataset (Table 2) they occupied $\sim 24MB$ and require $\sim 1G$ operations. An average of 171 variants per kernel were generated, ranging from 26 for `ij-ik-kj` to 296 for those with 6 dimensions.

Kernel	PTile	PTile+Intrin	PTile+Intrin +OpenMP	OpenMP Speed Up	OpenMP Efficiency
ij-ik-kj	0.08	1.23	10.34	8.43	0.28
ij-kil-lkj	0.46	1.88	7.08	3.77	0.13
ijk-il-jlk	0.32	0.75	3.11	4.12	0.14
ijk-ilk-jl	0.35	0.81	16.29	20.07	0.67
ijk-ilk-lj	0.28	0.78	14.81	18.98	0.63
ijk-ilmk-mjl	0.92	4.43	41.45	9.35	0.31
ijkl-imkn-njml	0.36	3.20	19.60	6.13	0.20
ijkl-imnk-njml	0.37	3.80	22.78	6.00	0.20
ijkl-minl-njmk	0.46	4.48	28.85	6.43	0.21
Geometric Mean	0.35	1.86	14.56	7.83	0.26

Table 1: Relative performance improvement over ICC using the input code for the standard dataset ($\sim 6MB$, $\sim 125M$ operations).

Table 1 shows the relative performance of our implementations to that obtained by ICC on a single core given the input code. The first column shows a significant drop in performance by enabling PTile without parallelization or vectorization. Inspection of the generated assembly shows that the introduction of complicated loop structures by PTile hinders ICC's ability to efficiently vectorize automatically. Introduction of intrinsics-based vectorized tiles provides a mean single core improvement of 86% over ICC. Finally, utilization of OpenMP to take full advantage of the MIC Architecture shows a mean improvement of $14.56\times$ over ICC single core performance. The right side of Table 1 shows the speed up of the code using PTile and vector intrinsics when OpenMP is enabled, and the efficiency of the 30 cores. Efficiency is defined as the ratio of the speed up to the number of processors and represents the utilizations of the cores.

Table 2 is structured the same as Table 1, except it is based on the large data set as described above. It is worth noting that the results for PTile are worse than with the standard dataset, implying that the heuristically chosen tile size is suboptimal for larger data, or the problem may require multiple levels of tiling. As such, the intrinsics and OpenMP versions are not as effective as in the standard dataset, however they are still improvements over compilation of the input code with ICC. Furthermore, there is a notable increase in the efficiency of parallelization with OpenMP, 32% vs. the standard dataset's 26%.

Due to the large number of cores available in KNF, and the relatively small number of iterations over loops in the case of tensor contractions with many dimensions, it is possible the performance of parallel code running with OpenMP is hindered by the ratio of processors to loop iterations. Merging outer parallel loops (possibly using OpenMP collapse) as well as modifications of the tile scheduling code could provide more loop iterations to be run in parallel

[1]`icc --version` reports `icc (ICC) Mainline 20120222`

Kernel	PTile	PTile+Intrin	PTile+Intrin +OpenMP	OpenMP Speed Up	OpenMP Efficiency
ij-ik-kj	0.01	1.18	11.93	10.08	0.34
ij-kil-lkj	0.42	1.67	6.00	3.60	0.12
ijk-il-jlk	0.43	1.00	4.77	4.77	0.16
ijk-ilk-jl	0.33	0.75	15.67	20.89	0.70
ijk-ilk-lj	0.29	0.78	16.07	20.66	0.69
ijk-ilmk-mjl	0.19	2.38	36.44	15.34	0.51
ijkl-imkn-njml	0.10	2.32	19.35	8.33	0.28
ijkl-imnk-njml	0.07	1.78	14.41	8.10	0.27
ijkl-minl-njmk	0.10	2.47	20.50	8.31	0.28
Geometric Mean	0.15	1.45	13.78	9.52	0.32

Table 2: Relative performance improvement over ICC using the input code for the large dataset ($\sim 24MB$, $\sim 1G$ operations).

and thus better utilize the available cores.

4 Related Work

Our previous work has shown that the selection of an efficient variant of vectorized L1 sized tiles can be performed at compile time using machine learning models [11], eliminating the need for auto-tuning. The models were trained on features extracted from the generated assembly code. Additionally, it has been shown that the technique is applicable to other codes than tensor contractions, specifically stencil computations are considered. The machine learning models are able to choose a variant at compile time that on average obtains 89% of the performance achieved with auto-tuning. Previous works have dealt with the issues surrounding alignment constraints with SIMD vectorization [4] [5] which could be used to address the KNF vector instruction set's lack of support for unaligned memory accesses.

While there exist simpler solutions to tiling perfectly nested loops such as tensor contractions [7], PTile is applicable to any irregular affine loop nest, and has been shown to produce comparable or better performance than other polyhedral based tiling algorithms [2] [8] [9].

Goto [6] showed that generating larger tile sizes to occupy L2 cache instead of L1 can be more efficient as one of the arrays remains only L2 resident and the cost to access it is amortized over the inner loop. Tuning of the tile sizes used in our work combined with the existing transformations could benefit performance in the same way.

5 Conclusion

Efficient utilization of modern hardware requires multiple transformations be applied to software. In this work we show that a source-to-source compiler can be developed to automatically apply the necessary transformations to fully utilize the Intel Many Integrated Core Architecture. By combining polyhedral tiling and parallelization tools with an aggressive vectorized tile code generator we achieve a mean improvement of 14.56× over existing compiler technology given the same input code on a Knights Ferry MIC Architecture Software Development Platform.

Future work on this subject will address the code generation limitations of this work. Development of a MIC-specific in-register transpose will enable the vectorization of non-unit-stride memory accesses, leading to the support of a wider set of tensor contractions. Machine learning models will enable compile time selection of efficient vectorized tiles from the large space of possible variants. Addressing alignment issues will enable vectorization in cases where the dimensions are not multiples of the vector width without modifying data structures (i.e. padding tensors). Finally, exploration over the space of possible tile sizes and multiple levels of tiling will allow for better exploitation of the memory hierarchy, either with compile time models or auto-tuning.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0926688.

References

[1] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 200–209, New York, NY, USA, 2010. ACM.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[3] T. Crawford and H. Schaefer III. An Introduction to Coupled Cluster Theory for Computational Chemists. In *Reviews in Computational Chemistry*, volume 14, pages 33–136. 2000.

[4] Alexandre Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, 2004.

[5] Liza Fireman, Erez Petrank, and Ayal Zaks. New algorithms for simd alignment. In *CC*, 2007.

[6] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[7] G. Goumas, M. Athanasaki, and N. Koziris. An efficient code generation technique for tiled iteration spaces. *Parallel and Distributed Systems, IEEE Transactions on*, 14(10):1021 – 1034, oct. 2003.

[8] Albert Hartono, Muthu Manikandan Baskaran, Cédric Bastoul, Albert Cohen, Sriram Krishnamoorthy, Boyana Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 147–157, New York, NY, USA, 2009. ACM.

[9] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 51:1–51:12, New York, NY, USA, 2007. ACM.

[10] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison. Model-driven simd code generation for a multi-resolution tensor kernel. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1058 –1067, may 2011.

[11] Kevin Stock, Louis-Noël Pouchet, and P. Sadayappan. Using machine learning to improve automatic vectorization. *ACM Trans. Archit. Code Optim.*, 8(4):50:1–50:23, January 2012.