Evaluation of DGEMM Implementation on Intel Xeon Phi Coprocessor

Pawel Gepner, Victor Gamayunov, David L. Fraser, Eric Houdard
Intel Corporation, Pipers Way, Swindon Wiltshire SN3 1RJ, United Kingdom
Email: {pawel.gepner, victor.gamayunov, david.l.fraser, eric.houdard}@intel.com

Ludovic Sauge, Damien Declat, Mathieu Dubois
Bull, 1, rue de Provence, 38432 Echirolles Cedex, France
Email: {ludovic.sauge, damien.declat, mathieu.dubois }@bull.net

***Abstract*— In this paper we will present a detailed study of implementing double-precision matrix-matrix multiplication (DGEMM) utilizing the Intel Xeon Phi Coprocessor. We discuss a DGEMM algorithm implementation running "natively" on the coprocessor, minimizing communication with the host CPU. We will run DGEMM across a range of matrix sizes natively as well using Intel Math Kernel Library. Our optimizations were designed to support maximal reuse of on-die cache, which significantly reduces transfer from GDDR. Finally we analyze the improvement of a classic matrix multiplication implementation based on Cauchy algorithm compared to the latest results achieved using the Intel Math Kernel Library DGEMM subroutine.**

***Index Terms*— Intel Xeon Phi coprocessor, high performance computing, matrix-matrix multiplication, performance measurements, Intel Math Kernel Library**

I. Introduction

Dense matrix operations are an important element in scientific and engineering computing applications. Today we have a significant amount of high performance libraries for dense matrix calculations. The Basic Linear Algebra Subprograms (BLAS) is a standard application-programming interface to execute basic linear algebra operations such as vector and matrix multiplication [2]. Historically the first BLAS program was released as a building block of LAPACK [1], which is a performance portable library for implementing dense linear algebra. Many hardware vendors also provide BLAS libraries tuned into their own architecture, i.e. Intel MKL, AMD CML or CUBLAS for the NVIDIA GPUs. To get the best performance all vendors optimize BLAS code specifically to the underlying hardware [1,5].

The intention of the this paper is to provide the best known method to run the classic Cauchy's based matrix multiplication code and evaluate the performance of the new coprocessor in comparison with Intel Math Kernel Library DGEMM subroutine optimized for Intel Xeon Phi Coprocessor.

To answer the research questions, we have organized the paper in the following way. Section 2 presents the platform architecture of the system as well as it introduces the microarchitecture of Intel Xeon Phi Coprocessor. In section 3 we characterize matrix multiplication algorithms. In section 4 we evaluate two types of algorithms for dense matrix-multiplications and analyze the performance considerations. In section 5 we draw conclusions and outline future work.

II. Platform Architecture And System Configuration

In our study we utilized two systems, the first one based on a dual socket Intel Xeon E5-2687W CPU processor with a single coprocessor card and the second one utilizing Intel Xeon E5-2680 CPU also with single coprocessor card. We have evaluated Intel Xeon Phi 3115A and Intel Xeon Phi SE10P cards respectively. The tested platform runs compiled versions of binaries, with Intel Xeon Phi support and optimized version of the Intel MKL library. Detailed hardware configuration is summarized in Table I. The new version of the Intel software tool kit has been also installed. The Intel Composer XE 2013 includes Intel Compiler 13.0, Intel Cilk Plus, Intel Math Kernel Library (MKL) 11.0, Intel Integrated Performance Primitives (IPP) 7.1 and the Intel Threading Building Blocks (TBB) 4.1. The new compiler and libraries offer simplified vectorization, guided auto parallelization support and high performance parallel optimizer. The new Intel Composer XE 2013 supports Intel Xeon Phi Coprocessor, new 3rd Generation Intel Core Processors (Intel microarchitecture code name: Ivy Bridge) and Intel the new 4th generation Intel Core Processors (microarchitecture code name: Haswell). All performance tools and libraries provide optimized parallel functions and data processing routines for high-performance applications and additionally contain several enhancements, including improved Intel AVX as well as AVX-2 support.

Manuscript received August 25, 2013; revised September 25, 2013; accepted November 15, 2013.

doi:10.4304/jcp.9.7.1566-1571

TABLE I.
CONFIGURATION DETAILS OF TESTED PLATFORMS

Host Platform	Bullx R425-E3	Intel "Crown Pass" SDP
CPU Type	Intel Xeon CPU E5-2680 (20M Cache, 2.7 GHz, 8.00 GT/s Intel QPI, 130 W TDP)	Intel Xeon CPU E5-2687W (20M Cache, 3.10 GHz, 8.00 GT/s Intel QPI, 150 W TDP)
Host OS	bullxlinux6.2	RedHat Enterprise Linux 6.3 x86_64
OS Version	2.6.32-279.9.1.el6.x86_64	
RAM	64GB of DDR3-1600 MHz	32GB of DDR3-1333 MHz
HDD	1x 1TB SATA	
Driver Version	4346-16	
MPSS Version	2.1.4346-16	
Coprocessor Details	Pre-production Intel Xeon Phi coprocessor 3115A: 57 cores, 1.1 GHz, 6 GB GDDR5 Memory@5 GT/s, 300 W TDP, B0 stepping, ECC enabled	Pre-production Intel Xeon Phi coprocessor SE10P: 61 cores, 1.1 GHz, 8 GB GDDR5 Memory@5.5 GT/s, 300 W TDP, B0 stepping, ECC enabled
Flash Version	2.1.01.0375	
uOS Version	2.6.34.11-g65c0cd9	

A. Intel Xeon Phi Coprocessor Characteristic

The Intel Xeon Phi coprocessor combines processing cores, caches, memory controllers, PCIe gen. 2 client logic interface, and a high bandwidth bidirectional ring interconnect. Each core has a private 64KB of L1 cache (32KB for Data and 32KB for Instructions) and also 512KB of L2 cache. These caches are fully coherent and implement the x86 memory order model. All L2 caches are kept fully coherent by a global-distributed tag directory. The L1 and L2 caches provide an aggregate bandwidth that is approximately 15 and 7 times, respectively, faster compared to the aggregate memory bandwidth. Effective utilization of the caches is key to achieving peak performance on the Intel Xeon Phi coprocessor.

The memory controllers and the PCIe client logic provide a direct interface to the 8GB of GDDR5 memory on the coprocessor and the PCIe bus. All these components are connected together by the ring interconnect. Each core in the Intel Xeon Phi coprocessor is designed to be power efficient while providing a high throughput for highly parallel workloads. The core uses a short in-order pipeline and is capable of supporting 4 threads in hardware. Based on silicon estimation the overhead to support legacy IA architecture is less than 2% of the core and L2 die area thus the cost of bringing the Intel Architecture legacy capability is very marginal [7].

An important component of the Intel Xeon Phi coprocessor's core is its vector processing unit (VPU). The VPU features a novel 512-bit SIMD instruction set, officially known as Intel Initial Many Core Instructions (Intel IMCI). The VPU can execute 16 single-precision (SP) or 8 double-precision (DP) instructions per cycle. The VPU also supports Fused Multiply-Add (FMA) instructions and hence can execute 32 SP or 16 DP floating-point operations per cycle. It also provides support for integers. The VPU also features an Extended Math Unit (EMU) that can execute single precision transcendental operations such as reciprocal, square root, exp2 and log2, thereby allowing these operations to be executed in a vector fashion with high bandwidth. The EMU operates by calculating polynomial approximations of these functions. The VPU also supports the gather and scatter instructions, which are simply non-unit stride vector memory accesses, directly in hardware. Thus for codes with sporadic or irregular access patterns, vector scatter and gather instructions help in keeping the code vectorized.

One of the major criteria during the development process was to make the vector processing unit highly power efficient for typical HPC workloads. A single operation can encode a great deal of work and does not incur energy costs associated with fetching, decoding, and retiring many instructions. However, several improvements were required to support such wide SIMD instructions. For example, a mask register was added to the VPU to allow per lane predicated execution. This helped in vectorizing short conditional branches, thereby improving the overall software pipelining efficiency.

The interconnection is implemented as a bidirectional ring. Each direction is comprised of three independent rings. The first, largest, and with the biggest performance degradation, due to its relatively remote location, is the data block ring. The data block ring is 64 bytes wide to support the high bandwidth requirement due to the large number of cores. The address ring is much smaller and is used to send read/write commands and memory addresses. Finally, the smallest ring and the least performance degradation prone is the acknowledgement ring, which sends flow control and coherence messages.

Other micro-architectural optimizations incorporated into the Intel Xeon Phi coprocessor include a 64-entry second-level Translation Lookaside Buffer (TLB), simultaneous data cache loads and stores, and 512KB L2 caches. Lastly, the Intel Xeon Phi coprocessor implements a 16-stream hardware prefetcher to improve the cache hits and provides higher bandwidth.

III. CHARACTERISTIC OF MATRIX MULTIPLICATIONS ALGORITHMS

Matrix multiplication defines

$$C=A*B, \quad (1)$$

where *A, B* and *C* are *m x k, k x n, m x n* matrices. A straightforward implementation of matrix multiplications is three nested loops. This Cauchy brute-force algorithm requires *m x n x k* multiplications and *m x n x (k-1)*

additions. For the square matrixes we have $m=n=k$ and number of operations is in order of n^3 [2].

More effective algorithms e.g. the Strassen algorithm reduces number of operations to order of $n^{2.8}$ in its place. This method is based on the concept of recursively partitioning a matrix into smaller blocks and reduces the most expensive multiplication operations [4,3,9].

The BLAS definition of DGEMM takes three matrices *A, B, C* and updates *C* with

$$beta*C + alpha*op(A)*op(B), \quad (2)$$

where *alpha* and *beta* are scalars and *op(A), op(B*) means that one can insert the transpose, conjugate transpose, or just the matrix as is. In our case we can ignore the *op(A), op(B), alpha, beta* and just focus on (1), based on the assumption that if we can do this operation efficiently we can incorporate a transpose or a scalar update equally as fast [8, 6].

A. Intel Xeon Phi Coprocessor And DGEMM

When we start code optimization on the Intel Xeon Phi coprocessor, the implementation of two fundamental tuning technics are critical for codes to take advantage of the Intel MIC architecture:

Vectorization: Loops (the most inner) need to use the hardware vector units.

Parallelization: At least two threads per core are needed for maximum instruction issue, and maximizing the pipeline utilization. Multiple threads imply a parallel program.

Applying this technique is fundamental, however to efficiently implement native DGEMM subroutine it is also necessary to organize the data structure in a MIC-friendly fashion and highly optimized basic kernels. Asynchronous Offload DGEMM requires choosing an appropriate size of matrix to maximize efficiency and minimize transfer latency as well as hide PCIe transfer overhead.

The simplest way to execute DGEMM on the Intel Xeon Phi Coprocessor is to use Intel MKL automatic offload. It enables users to take advantage of both, the host and the MIC coprocessor without any code changes. If the MIC coprocessor exists Intel MKL makes a smart decision on what operations are worth offloading, and what are the appropriate work partitions between the host and the coprocessor. The library also transparently handles data transfer and manages remote execution. If no MIC coprocessor is present, execution falls back to the host without any performance diadvantage.

In our scenario we discussed a DGEMM algorithm implementation running "natively" on the coprocessor only, minimizing communication with the host CPU, in this circumstance we can explicitly call pragma that becomes executed on MIC only. In this configuration, the MIC coprocessor can be viewed as a standalone (large) SMP node. The MIC coprocessor embeds a standard Linux kernel and exposes a virtual network interface to the host machine. The user can easily cross-compile any code on the host machine (using the Intel Compiler suite or GNU Compiler Collection-GCC cross-compiler), securely transferring the produced binary to the MIC and execute it through SSH.

As DGEMM has a large amount of data reuse to make this effective for Intel Xeon Phi, where cache is only limited to 512KB, tiling has been implemented as the major technique to reduce the bandwidth issue in communication with the GDDR5 memory on the coprocessor card. We can assume that, the bigger the tile size, the lower the bandwidth. The bigger the cache, the bigger we can make the tile size. The idea is to use as largest tile as possible. For Intel Xeon Phi GDDR tiling can reduce bandwidth down to 1.1 bytes/cycle in the best case scenario [7,10,11].

Choosing the right tile dimension is not a trivial task. In each step MIC multiplies matrix A tile by appropriate matrix B tile and the results are stored in matrix C tile. We need to keep them in L2 cache and below 512KB. In addition to controlling the size of matrixes, not to overwrite the L2 cache, we need to carefully acknowledge Translation Lookaside Buffer (TLB) limitation.

The Intel MIC Translation Lookaside Buffer has 64 entries for 4 KB pages. This means that it can map at most 256 KB of data. Whenever the amount of data exceeds this amount, a TLB miss will occur. TLB misses are very expensive. Large pages are used to reduce TLB misses. Think of this - 4KB page fits only 1024 floats. When matrix size is 1024 x 1024 or above every matrix column requires a separate TLB entry. So, we should be careful not to thrash the TLB, which might cost us resources, such as scanning over the output matrix multiple times.

IV. Evaluation Of Matrix Multiplications Algorithms

The intention of this paper is to evaluate different DGEMM implementations executed on Intel Xeon Phi coprocessor and finding the most effective one. We present our evaluation study for different implementation for different matrix sizes and various parameters and compiler options that impact overall performance. We have implemented a hand tuned Cauchy algorithm and tested it against different sizes of matrices then compared it to results achieved with the Intel MKL DGEMM subroutines. Applied methods are single and multithreaded.

In our study we have used *n x n* square matrices and all the elements of the matrices are generated as double precisions numbers.

We also tested and monitored usage of the system resources and we counted the number of executed floating-point operations and analyzed the resource usage.

Essentially in our study we have been evaluating two types of matrices sizes: a small one that is able to fit in to the L1 cache and a big one bigger than L1 can hold it.

We have also implemented Cauchy algorithms and performed several experiments with different values of the prefetch distance.

As already mentioned in Chapter 3, the Cauchy algorithm consists of 3 nested loops. In Figure 1, we present a possible implementation in C language. The code has been compiled using the latest version of the Intel ICC compiler incorporated into the Intel Composer XE suite toolbox. Version 2013.2.146 has been used for this work. If we consider Intel Xeon Phi we need to remember that software components are a critical part of

```
for(i=0;i<N;i++){
    for(j=0;j<N;j++)
        c[i*N+j]=a[i*N+0]*b[0*N+j];
    for(k=1;k<N;k++)
        for(j=0;j<N;j++)
            c[i*N+j]+=a[i*N+k]*b[k*N+j];
}
```

Figure 1. The basic framework of DGEMM routines

the performance chain. Because the Intel Xeon Phi is an in-order processor, optimizations and the scheduling of the instructions produced by the compiler are critical [7].

Basic performance values are estimated in native mode. In this mode, the code is (**cross**-) compiled on the host using the flag '**-mmic'**. The binary produced could be only executed on the co-processor. The binary either has to be located on a filesystem shared with co-processors, such as NFS, or copied (including any possible dependencies, e.g. shared libraries) to the co-processor. Then the user executes the code as he does on other standard Linux platform. Note that the MPSS (Many-core Platform Software Stack) contains a tool that automatically determines the library dependencies of your binaries, then copies them to the Intel Xeon Phi co-processor and executes the application.

Good practice is to control the assembly code produced by the compiler. To do this, we pass the compiler flag '**-S**' to produce an annotated assembler source code, identify the loops of interested, check the quality of the optimization proposed by the compiler, essentially vectorization, prefetching and floating point instruction involved.

In Figure 2 we show what the Cauchy inner loop looks like when generating compiler assembly output. Clearly the compiler does very good optimization work if the different parameters such as array bounds are known at compiling time. Vectorization, loop unrolling and instruction scheduling look very good in this case. Basically it is unrolled by 8 and there are 8 FMAs, so we get exactly 64 double precision flops per one iteration of the inner loop (Each FMA operating on 512 bit wide vectors). Considering the algorithm presented in Figure 1, a total of N^3 iterations are needed (N^2 for matrix initialization and $N^2(N-1)$ for matrix update). Concerning the memory operations: *'b'* elements are read N^3 times; the same element *'a[i*N + k]'* is used by all iteration of the innermost loop. Therefore, *'a'* elements are read *N(2N-1)* times. Because due to the cache coherency protocol writing a memory word requiring a prior reading (RfO or Read for Ownership operation), so *'c'* elements are both written and read N^3 times. Therefore, a total of $3N^3+2N^2-N$ memory operations are performed.

If streaming stores (non-temporal write operation) are used, no prior reading of the cache line is required. In this case, all updated elements are collected in the write-combining buffer and sent to memory when full. There are two advantages of this technique: first, it reduces the memory traffic, from 2 ops (1 read + 1 write) to only 1 op (1 write). The second advantage is that, these operations by-pass the last level cache (L2 on Xeon Phi and L3 on Sandy Bridge Processor) and does not pollute it. On the other hand, the main disadvantage is that in this case, the cache coherency must be handled by software, directly by the compiler or by the programmer. In our case, the number of memory operations could be reduced to $2N^3+N^2$.

Concerning the total number of floating point operations: during the matrix initialization (first nested loops), N^2 multiplications are performed. Subsequent matrix updates (next 3 nested loops), involve $N^2(N-1)$ both multiplications and additions. Note that these latest operations could be combined into a single FMA instruction (fused multiply-add operation). Therefore a total of $2N^3-N^2$ floating-point operations are required N^3 scalar multiplications and $N^2(N-1)$ scalar addition.

Because of the micro-architecture of the Intel Xeon Phi processor, 2 instructions running on the same thread could not be executed back to back in the same cycle. Therefore, throughput and latency of the instruction is at least 2 cycles. This is why it requires at least using 2 threads on the same core to get the best performance.

```
vmovapd   1024+b(%r11,%rdx,8), %zmm1                    #34.62 c1"
vmovapd   1088+b(%r11,%rdx,8), %zmm2                    #34.62 c5"
vmovapd   1152+b(%r11,%rdx,8), %zmm3                    #34.62 c9"
vmovapd   1216+b(%r11,%rdx,8), %zmm4                    #34.62 c13"
vmovapd   1280+b(%r11,%rdx,8), %zmm5                    #34.62 c17"
vmovapd   1344+b(%r11,%rdx,8), %zmm6                    #34.62 c21"
vmovapd   1408+b(%r11,%rdx,8), %zmm7                    #34.62 c25"
vmovapd   1472+b(%r11,%rdx,8), %zmm8                    #34.62 c29"
vfmadd213pd c(%rsi,%rdx,8), %zmm0, %zmm1                #34.33 c33"
vfmadd213pd 64+c(%rsi,%rdx,8), %zmm0, %zmm2             #34.33 c37"
vfmadd213pd 128+c(%rsi,%rdx,8), %zmm0, %zmm3            #34.33 c41"
vfmadd213pd 192+c(%rsi,%rdx,8), %zmm0, %zmm4            #34.33 c45"
vfmadd213pd 256+c(%rsi,%rdx,8), %zmm0, %zmm5            #34.33 c49"
vfmadd213pd 320+c(%rsi,%rdx,8), %zmm0, %zmm6            #34.33 c53"
vfmadd213pd 384+c(%rsi,%rdx,8), %zmm0, %zmm7            #34.33 c57"
vfmadd213pd 448+c(%rsi,%rdx,8), %zmm0, %zmm8            #34.33 c61"
vmovapd   %zmm1, c(%rsi,%rdx,8)                         #34.33 c65"
vmovapd   %zmm2, 64+c(%rsi,%rdx,8)                      #34.33 c69"
vmovapd   %zmm3, 128+c(%rsi,%rdx,8)                     #34.33 c73"
vmovapd   %zmm4, 192+c(%rsi,%rdx,8)                     #34.33 c77"
vmovapd   %zmm5, 256+c(%rsi,%rdx,8)                     #34.33 c81"
vmovapd   %zmm6, 320+c(%rsi,%rdx,8)                     #34.33 c85"
vmovapd   %zmm7, 384+c(%rsi,%rdx,8)                     #34.33 c89"
vmovapd   %zmm8, 448+c(%rsi,%rdx,8)                     #34.33 c93"
addq      $64, %rdx                                     #33.25 c97"
cmpq      $128, %rdx                                    #33.25 c101
jb        ..B2.8        # Prob 99%                      #33.25 c101
```

Figure 2. The assembly code of the most inner loop of Cauchy algorithms

Exposing thread parallelism is critical to achieving good performance therefore, we parallelize the code using standard unix/linux pthreads. As the Intel Xeon Phi is an x86 processor, we can use programming methods, that we usually employ with standard Intel Xeon CPU. As we

already mentioned, porting a code to Intel Xeon Phi in native mode is straightforward and requires minimal porting effort, generally reduced to recompiling the application using **'-mmic'** flag.

Different memory allocation strategies have been tested (static, dynamic, using or not huge pages). To get the best performance as possible, all arrays have been aligned. In the case of Intel Xeon Phi architecture, 64-bytes alignment (because of the width of the vector) is recommended. Aligning array is generally not sufficient: it is necessary to tell the compiler that it has to deal with aligned arrays helping it to generate more efficient code. This could be done by inserting the **#pragma** vector aligned into code

During our evaluation we have been testing usage of prefetching (both software and hardware) and their implications to achieved performance. Prefetching is a well-known technique to hide memory access latency, speculatively loading data in the cache hierarchy likely to be requested by processor to reduce the impact of cache misses. Prefetching could be automatic and hardwired in processor (hardware prefetchers). The current Intel Xeon Phi architecture includes a hardware prefetcher (HWP) from GDDR into L2. It is activated by default and has a modified autonomous prefetching unit design that originated with the Intel Pentium 4 family [7]. Allocation occurs upon a demand L2 miss to a new 4K page. Once a stream direction (forward or backward) is detected, the hardware prefetcher issues up to four-cache line prefetch requests, which are the same type as the access. Unlike a standard Intel Xeon CPU like Sandy Bridge, this hardware prefetcher is unable to deal with stridden memory access. Only sequential memory access can be handled. In case of stridden memory access, software prefetching optimization techniques must be considered [7].

Intel Xeon Phi coprocessor supports a variety of software prefetching instructions. These instructions provide the ability to prefetch data to a specified cache level (L1 or L2). Two different types of software prefetches can be considered: software prefetch automatically generated by the compiler and manual or assisted prefetch by the programmer. The Intel Compiler generates appropriate software prefetch instructions for the compiled code with **–O2** and above optimization level. It implies first prefetch to the L2 cache followed by another prefetch to the L1 cache. Note that L1 and L2 prefetch instructions can pair together (in either U- or V-pipe) and be issued in the same processor cycle. In the same way, a prefetch instruction issued in V-Pipe can be executed concurrently and paired with a vector instructions issued in the U-Pipe. These features help to reduce and partially eliminate the impact of the prefetch instructions [7]. Prefetch distances are computed by the compiler based on the amount of work inside the loop. However, the compiler's automatic prefetching capabilities might be limited. In this case, the programmer has the possibility to control manually and fine-tune prefetches using specific compiler flags, inserting pragma or using intrinsics **mm_prefetch()** in C/C++ and **MM_PREFETCH** in Fortran.

As we can see on Figure 3, we produced different static versions of the code, depending on the size of the

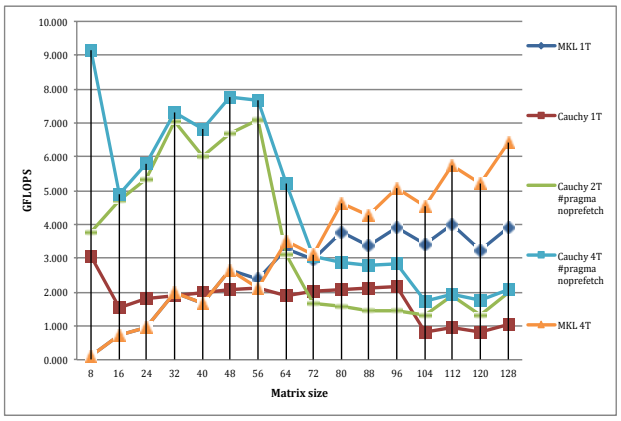

Figure 3. Results of DGEMM for small matrices up to 128x128.

problem but also as we already mentioned, different memory allocation strategies (static, dynamic, using or not huge pages), using one, two or four threads per core. We contrast these results against version compiled with the Intel MKL DGEMM subroutine. Maximum size of the problem we have tested in this scenario was 128x128. As we can see, as long as the matrix fits into L1 cache of the Intel Xeon Phi (32 KB), that the performance of our hand tuned Cauchy implementation outperforms the results achieved by the Intel MKL subroutine. The best results up to the size of matrices 64x64 we achieved when we runing the 4 threads per core version of Cauchy followed by two threads version of Cauchy algorithm. For these pthreads versions we split the loop into a quarter or half respectively. Each extra thread is spinning and waiting for the signal to start the loop. This way the overhead is quite small. The pragma with prefetch disabled for k loop is the faster single thread version. The MKL version for one, two and four threads gives the same results and as long as the matrix fits into the L1 cache, it is significantly below what our hand tuned version can achieve. Even 4 threads version of Intel MKL up to matrix size 56x56 performs significantly worse than the threaded version of Cauchy implementation.

In our case, for small matrices fitting the cache, software prefetching is inefficient or worst, counterproductive and degrades the performance. Compiler prefetching has been disabled using the compiler flag **-no-opt-prefetch** or **-opt-prefetch=0** (explaining the lack of assembly instructions **vprefetch0** and **vprefetch1** in the assembly code presented in Figure 2). Other compiler options are available to disable only L1 or L2 prefetch instructions, or to fine-tune the prefetch distances. The disadvantage of this method of course is that it is global. Compiler prefetches can also be disabled for specific loops, inserting just before the loop the pragma **#pragma noprefetch** in C/C++ language.

For the large matrices we plot the results in the Figure 4 and clearly we can see that the best results are achieved

for Intel MKL DGEMM subroutine and difference is significant against what we can accomplish with Cauchy algorithm implementation. For large matrices, we get better performance when we pad the different arrays

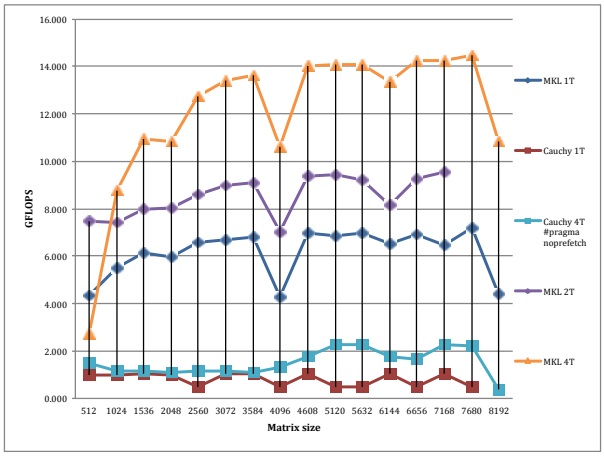

Figure 4. Results of DGEMM for matrices up to 8192x8192.

representing the three different matrices *A*, *B* and *C*. Array padding improves performance-avoiding cache set conflict. Figure 4 clearly demonstrates such effects at matrix size of 8192 or 4096 where we used no padding on arrays.

Both L1 and L2 cache are 8-ways cache as described earlier. So in worst case, both L1 and L2 caches may be reduced only to 512 bytes in size in case of set conflict. Playing with different settings of prefetchers, allocation of matrices and huge pages enabling does not change the situation and Intel MKL DGEMM subroutines provide the best results far ahead of what we can achieve by our hand tuned Cauchy implementations.

V. Conclusions

This paper examined two algorithms for dense matrix-multiplications running natively on the Intel Xeon Phi Coprocessor. In the evaluation we have tested a classic brute-force algorithm and compared it against the Intel Math Kernel Library DGEMM subroutine.

The completed results clearly show that Intel Math Kernel Library, for larger than 128x128 matrices provides 2 to 12 times better performance versus brute-force algorithm, but for the small matrices (up to 64x64) fitting into Intel Xeon Phi Coprocessor L1 cache, the Cauchy algorithm implementation has better performance by 50% up to even 900%.

The difference is greater for Intel MKL DGEMM subroutines when the size of matrix increases. The scalability of performance increases accordingly to the number of incorporated threads and the utilization of the resources. The Intel ICC compiler performs excellent optimization work in the case of vectorization but the results achieved by Intel MKL DGEMM subroutines surpass this.

Generally we advise to use the 4-threaded version of the Cauchy algorithm for small matrices below the size of 72x72 and use Intel Math Kernel Library DGEMM subroutine for all other cases where the size of the matrix is bigger than the Intel Xeon Phi Coprocessor L1 cache can hold.

It is also important to mention that for small matrices the performance achieved by the Intel Xeon Phi Coprocessor are worse than results obtained on Intel Xeon E5-2687W CPU by 200%-300% running a brute-force algorithm or Intel MKL version of the DGEMM.

Acknowledgments

We gratefully acknowledge the help and support provided by Jamie Wilcox from Intel EMEA Technical Marketing HPC Lab and Johann Peyrard and Sylvain Cohard from Bull Extreme Computing Applications & Performance Team.

References

[1] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. "Self-adapting linear algebra algorithms and software." *In Proceedings of the IEEE*, vol. 93, pp 293–312, 2005.

[2] J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. "A set of level 3 basic linear algebra subprograms". *ACM Trans. Math. Softw 16*, pp.1–17, March 1990.

[3] P.Gepner, V Gamayunov and D. L. Fraser. "Early performance evaluation of AVX for HPC". *In Proceedings of International Conference on Computational Science, ICCS 2011, Singapur*, June 2011

[4] P.Gepner, V Gamayunov and D. L. Fraser. "Evaluation of Executing DGEMM Algorithms on modern MultiCore CPU". *In Proceedings of The Parallel and Distributed Computing and Systems 2011 Conference*, Dallas, December 2011

[5] K. Goto and R. A. v. d. Geijn. "Anatomy of high-performance matrix multiplication". *ACM Trans. Math. Softw*, pp. 1–25, May 2008.

[6] G. Henry. "Optimize for Intel AVX Using Intel Math Kernel Library's Basic Linear Algebra Subprograms (BLAS) with DGEMM Routine". *Intel Software Network*, July 2009

[7] J. Jeffers and J Reinders, "Intel Xeon Phi Coprocessor High Performance Programming", *publised by Morgan Kaufmann ia an imprint of Elsevier*.

[8] P. Kopta, M. Kulczewski, K. Kurowski, T. Piontek, P. Gepner, M. Puchalski and J. Komasa, "Parallel application benchmarks and performance evaluation of the Intel Xeon 7500 family processors", *Procedia Computer Science 4 (2011)* pp. 372–381.

[9] V. Strassen. "Gaussian elimination is not optimal". *Numer. Math, 13*, pp. 354–356, 1969.

[10] M. Krotkiewski and M. Dabrowski. "Parallel symmetric sparse matrix-vector product on scalar multi-core CPUs." *Parallel Comput., 36(4), pp.* 181–198, Apr. 2010.

[11] N. Bell and M. Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". *In Proc. High Performance Computing Networking, Storage and Analysis, SC '09*, pp 18:1–18:11, 2009.