Integrating Cache Oblivious Approach with Modern Processor Architecture: The Case of Floyd-Warshall Algorithm

Toshio Endo
endo@is.titech.ac.jp
GSIC, Tokyo Institute of Technology / RWBC-OIL, AIST

ABSTRACT

In order to implement algorithms on processors with deep cache hierarchy, the cache oblivious approach, which is based on recursive divide and conquer, is considered to be promising. This paper focuses on single-node implementation of Floyd-Warshall (FW) algorithm, which is an important graph computation kernel. For higher performance, another facility of modern processors, SIMD instructions need to be integrated to recursive approach efficiently. This paper describes a methodology to construct recursive implementations that takes architecture with SIMD and multi-core into account while harnessing cache. The experiment shows our FW implementation exhibits around 1.1 TFlops on a dual-socket SkyLake machine and 700 GFlops on a Xeon Phi machine, both of which have AVX512 SIMD ISA.

CCS CONCEPTS

• **Computing methodologies → Parallel algorithms**; • **Computer systems organization →** *Parallel architectures.*

KEYWORDS

graph algorithm, memory hierarchy, cache oblivious, SIMD

ACM Reference Format:
Toshio Endo. 2020. Integrating Cache Oblivious Approach with Modern Processor Architecture: The Case of Floyd-Warshall Algorithm. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020), January 15–17, 2020, Fukuoka, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3368474.3368477

1 INTRODUCTION

In recent modern computer architecture, memory access tends to become performance bottleneck in wide range of applications. Due to this problem, often called "memory wall" problem [1], it is the key to harness cache memory hierarchy in order to alleviate memory access costs. In this context, one of standard techniques is *cache blocking*, with which each processor core performs computation on a smaller block that fits within cache size. The technique has been widely used to accelerate dense linear algebra[2–4], sparse linear algebra[5, 6], stencil computation[7–9], graph algorithms[10, 11] and so on. In order to realize efficient cache blocking, the implementation should take care of capacity of each cache level, which can be different among processor products. Also different processors may have different number of cache levels, while they support the same instruction set architecture (ISA), such as Intel SkyLake Xeon with three level cache and Xeon Phi with two level cache[12].

As more architecture independent direction, *cache oblivious* approach has been proposed[13]. The approach is based on recursive divide and conquer strategy and can make the working set size fit each cache level automatically without specifying cache sizes as architecture parameters. Theoretical analyses on the amount of cache misses have been made for several algorithms[14–17]. While this approach has been successful both in theory and implementation on modern architecture with deeper cache/memory hierarchy, not many studies on the relationship with SIMD parallelism, another feature in modern processors, have been done. Without using SIMD parallelism, we cannot bring out high performance of processors.

This paper focuses on the all-pairs shortest path (APSP) problem, which is an important kernel of graph analysis area. And we take the *Floyd-Warshall* (FW) algorithm, one of well-known algorithms for APSP and describe a single-node implementation on top of multi-core processors that have AVX512 SIMD ISA. The FW algorithm has been shown to achieve high performance by using SIMD parallelism, it significantly suffers from access costs to main memory[11]. Our implementation harnesses two types of parallelism, SIMD parallelism and multi-core parallelism, while mitigating access to main memory largely by using cache oblivious approach. The basic idea is that we use high performance kernels tuned with SIMD instructions as the basis of the recursive call. Multi-core parallelism is also harnessed naturally by embedding task creation in the recursion. Due to this design, the implementation is not fully architecture independent despite the original idea of the cache oblivious approach. The experiment shows our implementation exhibits around 1.1 TFlops on a dual-socket Skylake Xeon machine and 700 GFlops on a Xeon Phi machine (single precision).

The main contributions of this paper are as follows:

- A method to implement high performance algorithm on modern processors by combining cache oblivious approach and SIMD parallelism is described.
- As a case study, an implementation of FW algorithm is described. The performance reaches 22% of the peak performance (44% if FMADD is not considered) with 32 SkyLake cores. A developing version of the implementation is publicly available at `https://github.com/toshioendo/hoalgos`.

```
1: procedure FW($D$)
2:     for $k = 0, \ldots, N-1$ do
3:         for $i = 0, \ldots, N-1$ do
4:             for $j = 0, \ldots, N-1$ do
5:                 if $D[i,j] > D[i,k] + D[k,j]$ then
6:                     $D[i,j] = D[i,k] + D[k,j]$
```

Figure 1: A simple FW algorithm

2 BACKGROUND AND RELATED WORK

2.1 Floyd-Warshall Algorithm

The Floyd-Warshall (FW) algorithm computes the shortest paths between all pairs of N vertexes of a given directed graph in compute complexity of $O(N^3)$. The input of the algorithm is a dense $N \times N$ matrix D, where $D_{i,j}$ is the length of an edge from i to j. If there is no edge from i to j, $D_{i,j}$ is set to an infinity (often implemented as a very large positive number).

Figure 1 shows a simple algorithm of FW. After its execution, the matrix D contains the shortest path length of each vertex pair. This algorithm, which is based on triply nested loop, has a similar structure to dense matrix-matrix multiplication (MM) algorithm. However, since the computation in this algorithm is done in-place, data dependency that does not appear in MM has to be considered in blocked or divide-and-conquer implementation.

Considering such dependency, blocked algorithms[10] and recursive divide-and-conquer algorithms[17–19] have been proposed. Figure 2 is an overview of recursive implementation. The algorithm starts from FW function that calls FW-REC function with specifying the input matrix D as three parameters. In FW-REC, three matrices are divided (lines 11 to 13) as shown in Figure 3, and FW-REC is called recursively for eight times (lines 14 to 21). In actual implementation, matrix division does not trigger copying of contents; instead, indices of matrices are used to locate partial matrices to be computed. If the matrices are "sufficiently" small, FW-BASE is called as the base case to stop recursion. The function is similar to that in Figure 1, but input matrices A, B may be different from the output C, while A and/or B may be same as C (alias cases).

Park et al.[17] have theoretically studied memory access traffic with the cache size C and shown that the divide-and-conquer algorithm achieves the traffic amount of $\Omega(N^3/\sqrt{C})$, which is asymptotically optimal. This is achieved without knowledge of architecture parameters such as C, thus this algorithm is cache-oblivious. Also several reports have shown this approach works efficiently through experimentation. On the other hand, not many studies on the relationship with SIMD instructions, which are keys to harness plenty of ALUs equipped to modern processors.

2.2 Target Processors with AVX-512 SIMD Instructions

In this paper, we mainly use an Intel Xeon SkyLake machine and an Xeon Phi KNL machine, whose specifications are shown in Table1. Both machines support Intel's recent SIMD instruction set, called AVX-512[20], the successor of SSE/AVX/AVX2. Especially, this paper focuses on `float` (FP32) data type and `avx512f` subset is used.

```
1: procedure FW-BASE($A, B, C$)
2:     for $k = 0, \ldots, BS-1$ do
3:         for $i = 0, \ldots, BS-1$ do
4:             for $j = 0, \ldots, BS-1$ do
5:                 if $C[i,j] > A[i,k] + B[k,j]$ then
6:                     $C[i,j] = A[i,k] + B[k,j]$
7: procedure FW-REC($A, B, C$)
8:     if $A, B, C$ is smaller than a threshold then
9:         FW-BASE($A, B, C$)
10:    else
11:        Divide $A$ into $A_{00}, A_{01}, A_{10}, A_{11}$
12:        Divide $B$ into $B_{00}, B_{01}, B_{10}, B_{11}$
13:        Divide $C$ into $C_{00}, C_{01}, C_{10}, C_{11}$
14:        FW-REC($A_{00}, B_{00}, C_{00}$)
15:        FW-REC($A_{00}, B_{01}, C_{01}$)
16:        FW-REC($A_{10}, B_{00}, C_{10}$)
17:        FW-REC($A_{10}, B_{01}, C_{11}$)
18:        FW-REC($A_{11}, B_{11}, C_{11}$)
19:        FW-REC($A_{11}, B_{10}, C_{10}$)
20:        FW-REC($A_{01}, B_{11}, C_{01}$)
21:        FW-REC($A_{01}, B_{10}, C_{00}$)
22: procedure FW($D$)
23:     FW-REC($D, D, D$)
```

Figure 2: Overview of recursive divide-and-conquer FW algorithm. BS in FW-BASE is the small problem size in base cases.

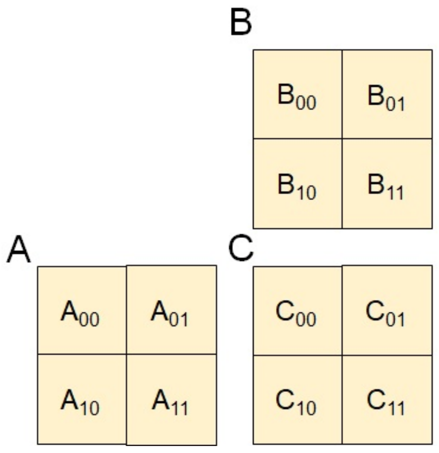

Figure 3: Division of matrices in FW-REC function.

With AVX-512, each core has 32 512-bit registers called ZMM registers, while the previous AVX2 provides 16 256-bit registers. Users can pack 16 float (FP32) elements or 8 double (FP64) elements into a single ZMM register. AVX-512 provides instruction set to manipulate the ZMM registers. For example, the `vaddpd` instruction, which is expressed by the `_m512_add_ps(a, b)` intrinsic function in C/C++, executes 16 summation operations to each float element of packed `a` and `b`.

Table 1: The target machines. In the evaluation in the paper, turbo boost is off.

	SkyLake machine	KNL machine
# of CPUs/machine	2	1
CPU	Xeon Gold 6140 (SkyLake)	Xeon Phi 7210 (Knights Landing)
# of cores/CPU	18	64
Clock (base)	2.3GHz	1.3GHz
L1D cache	32KiB/core	32KiB/core
L2 cache	1MiB/core	1MiB/2-cores
L3 cache	24.75MiB/CPU	(none)
Supported SIMD	avx512f, avx512dq, avx2, etc.	avx512f, avx2, etc.
Peak perf/core		
- double (FP64)	73.6GFlops	41.6GFlops
- float (FP32)	147.2GFlops	83.2GFlops
Peak perf/CPU		
- double (FP64)	1326GFlops	2662GFlops
- float (FP32)	2652GFlops	5324GFlops
MCDRAM Memory	(none)	8channels
Capacity		16GiB
Bandwidth		~500GB/s
DDR4 Memory	DDR4-2666 6ch × 2	DDR4-2400 6ch
Capacity	192GiB	192GiB
Bandwidth	256GB/s	115GB/s
OS	CentOS 7.6	CentOS 7.3
Compiler	Intel 19.0.2	Intel 19.0.2

With SIMD instructions, the FP32 peak performance of the SkyLake machine in Table1 is calculated as follows. Here, we assume the clock frequency is fixed at the base clock, 2.3GHz[1]. While each typical SIMD operation can execute 16 FP32 operations, a fused multiply-add (FMADD) operation such as _mm512_fmadd_ps executes 16×2 (=mul+add) operations. The later is usually counted to calculate peak performance. Also each core has two AVX-512 units. In total, the core peak performance is calculated as 16×2×2 2.3GHz = 147.2GFlops. Thus peak with two SkyLake CPUs (18 × 2 cores) is 5.3TFlops. Through the similar discussion, FP16 peak performance of KNL machine is 83.2GFlops per core and 5.32TFlops totally. However, if the target software cannot utilize FMADD operations, the upper limit of performance is the half of the above theoretical number. FW algorithm with SIMD extension described below falls under this category.

There is a significant difference between the SkyLake machine and the KNL machine in memory hierarchy. First, the former has 3 level caches, while the latter has 2 level caches. On the other hand, the KNL machine has two types of main memory layer; a common DDR4-DRAM layer and an MCDRAM layer with higher bandwidth. Xeon Phi provides several configuration modes for main memory; this paper uses the "flat" mode, where DDR4 and MCDRAM are logically flat and each layer corresponds to a NUMA node.

[1]According to Intel's white paper, clock frequency when AVX-512 instructions are running is lower than the base clock frequency[21]. In this discussion, this slow down is ignored for simplicity

```
1:  procedure FW-BASE-SIMD($A, B, C$)
2:      for $k = 0, \ldots, BS - 1$ do
3:          for $i = 0, \ldots, BS - 1$ by 16 do
4:              $a$ = _mm512_loadu_ps(&$A[i, k]$)
5:              for $j = 0, \ldots, BS - 1$ do
6:                  $b$ = _mm512_set1_ps($B[k, j]$)
7:                  $c$ = _mm512_loadu_ps(&$C[i, j]$)
8:                  $sum$ =_mm512_add_ps($a, b$)
9:                  $mask$ =_mm512_cmp_ps_mask
10:                         ($sum, c, \_CMP\_LT\_OQ$)
11:                 _mm512_mask_storeu_ps(&$C[i, j], mask, sum$)
```

Figure 4: Pseudo-code of a kernel function for a block size of BS with AVX-512 operations by Rucci et al.[11] Modified for explanation.

2.3 FW Implementation with AVX-512

Using AVX-512 instructions, Rucci et al. has described a blocked, non-recursive implementation of FW algorithm [11] . The pseudo code of the kernel for a block size of BS is shown in Figure 4, which is slightly modified for explanation. In this paper, the matrix is aligned in column-major order. As described in above, since the input blocks A, B, C may be aliased, k-loop is at outermost to preserve dependency.

The basic idea of this kernel is to compute 16 consecutive elements in C, which are $C_{i,j}, C_{i+1,j}, \ldots, C_{i+15,j}$ at once by using values of $A_{i,k}, A_{i+1,k}, \ldots, A_{i+15,k}$ and $B_{k,j}$. In lines 4 and 7, 16 elements in A and C are loaded from memory and packed into SIMD registers, a and c, respectively. For B, a single element $B_{k,j}$ is used for 16 operations. For this reason, this element is broadcast to all 16 elements in b by _mm512_set1_ps operation (line 5). Then sum, summation of a and b, is computed and then compared with c by using a "mask' register. Now $a[s]$ $(0 \leq s \leq 15)$ denotes s-th element of a single SIMD register a. The operation in lines 9–10 means $mask[s] = 1 if sum[s] < c[s]$ for all s. In line 11, elements of C on memory is updated by $sum[s]$ if $mask[s] == 1$.

The number of floating operations of this kernel is $2BS^3$, counting add and compare operations. For the entire FW algorithm, that is $2N^3$.

Rucci et al. have evaluated the performance on a Xeon Phi KNL machine and achieved up to 338GFlops with MCDRAM memory.

3 RECURSIVE FW IMPLEMENTATION WITH SIMD

3.1 Single-core Implementation

This section describes our Floyd-Warshall implementation based on cache-oblivious approach for a single core. Our starting point is a combination of recursive call described in Figure 2 and an AVX-512 based kernel in Figure 4. A simple combination, however, still suffers from slow down caused by property of machine architecture. Hereafter, we describe several techniques to improve the performance. Many of them are well known in the area of dense linear algebra software, and we will demonstrate that those techniques have significant impacts on the divide-and-conquer FW algorithm.

3.1.1 *Block-Aligned Recursive Division and Block Size.*

In the original recursive algorithm, the input matrix size N is divided into half repeatedly until the size gets smaller than a threshold. Then the small block is computed by the base function. The division size there, BS in functions in Figures 2 and 4, should be a multiple of 16, which is the number of float elements in a SIMD register. In order to achieve this, we keep the division size to be a multiple of 16 in every recursive call, as far as possible.

The value of BS is configured to be 64, unless otherwise noted.

3.1.2 *Using Block Data Layout.*

Algorithms described so far assume that the input matrix is in the column-major format. Using this format, algorithm suffers slow down by the following reasons. First, even if the algorithm processes smaller blocks than cache capacity, it may suffer from conflict cache miss costs when data elements in each block are scattered. Secondly, if N is indivisible by 16, memory accesses in lines 4, 7 and 11 in Figure 4 are unaligned accesses, significantly slower than aligned accesses.

To alleviate those memory access costs, we transform the matrix into *block data layout*[22, 23]. This layout assumes that the block size BS is predefined, and $BS \times BS$ elements in each block are placed contiguously on memory. When each block is sufficiently smaller than cache size, conflict misses are minimized. Also when BS is a multiple of 16 ($BS = 64$ in this paper), unaligned accesses are avoided. When this technique is adopted, we suffer from transform cost of $O(N^2)$, but it is asymptotically smaller than FW computation cost of $O(N^3)$.

3.1.3 *Second Kernel with Register Blocking.*

In FW, k-loop must be at the outermost in order to preserve data dependency. Thus the SIMD kernel in Figure 4 has the same structure, and the contents of a SIMD register c is written back to the matrix in each iteration. These access are expected to be cache-hit accesses, however, cache access costs are larger than costs of register manipulation. This kernel, which writes back c to the matrix frequently, is too pessimistic when A, B, C are not aliased and different from each other.

Improving performance of those "non-aliased" cases is important, since such cases are major in the entire FW algorithm. The number of kernel calls for "aliased" cases is only $O((N/BS)^2)$, out of $O((N/BS)^3)$ total kernel calls.

For non-aliased cases, we have implemented another kernel function that minimizes load/store operations as shown in Figure 5. Since there is no possibility for aliases, we have more choices for the order of loops. Here there are i, j-loops outside of k-loop. In lines 4-32, we focus on a 16×16 small block in C, smaller than a $BS \times BS$ block. Here we use 16 SIMD registers, denoted as $c0, c1, \cdots, c15$ in the code. They are used to accumulate temporary results and this technique is a kind of *register blocking* [24, 25].

First, all elements in $c0, c1, \cdots, c15$ are initialized by infinity (a very large positive value in the implementation). Then for each k, a local result that corresponds to $C_{i+ii,j+jj}$ is accumulated to the ii-th element of cjj (lines 8-21). After k-loop finishes, values of $c0, c1, \cdots, c15$ are reflected to the matrix C. Here occurs comparison between SIMD registers again (lines 22-32). This way, this kernel writes back results to C much less frequently than that in Figure 4.

```
 1: procedure FW-Base-RegBlock($A, B, C$)
 2:     for $i = 0, \ldots, BS - 1$ by 16 do
 3:         for $j = 0, \ldots, BS - 1$ by 16 do
 4:             $c0$ = _mm512_set1_ps ($\infty$)
 5:             $\cdots$
 6:             $c15$ = _mm512_set1_ps ($\infty$)
 7:             for $k = 0, \ldots, BS - 1$ do
 8:                 $a$ = _mm512_loadu_ps (&$A[i, k]$)
 9:                 // for $c0$
10:                 $b$ = _mm512_set1_ps($B[k, j + 0]$)
11:                 $sum$ =_mm512_add_ps($a, b$)
12:                 $mask$ =_mm512_cmp_ps_mask
13:                         ($sum, c0, \_CMP\_LT\_OQ$)
14:                 $c0$ = _mm512_mask_blend_ps($mask, c0, sum$)
15:                 $\cdots$
16:                 // for $c15$
17:                 $b$ = _mm512_set1_ps($B[k, j + 15]$)
18:                 $sum$ =_mm512_add_ps($a, b$)
19:                 $mask$ =_mm512_cmp_ps_mask
20:                         ($sum, c15, \_CMP\_LT\_OQ$)
21:                 $c15$ = _mm512_mask_blend_ps($mask, c15, sum$)
22:             // for $c0$
23:             $tmpc$ = _mm512_loadu_ps(&$C[i, j + 0]$)
24:             $mask$ =_mm512_cmp_ps_mask
25:                     ($c0, tmpc, \_CMP\_LT\_OQ$)
26:             _mm512_mask_storeu_ps(&$C[i, j + 0], mask, c0$)
27:             $\cdots$
28:             // for $c15$
29:             $tmpc$ = _mm512_loadu_ps(&$C[i, j + 15]$)
30:             $mask$ =_mm512_cmp_ps_mask
31:                     ($c15, tmpc, \_CMP\_LT\_OQ$)
32:             _mm512_mask_storeu_ps(&$C[i, j + 15], mask, c15$)
```

Figure 5: A kernel function with AVX-512 operations that have fewer load/store memory accesses. This works correctly only if A, B, C are not aliased (different from each other).

We used the small block size of 16×16. While the first dimension comes from the number of float elements, the second dimension, which corresponds to the number of accumulators, can be tuned. However, we should not choose too large number since a single core has only 32 SIMD registers.

3.2 Multi-core Implementation

This section extends the divide-and-conquer FW implementation describe above to harness multi-core parallelism. We can parallelize this algorithm by using parallel task creation. For this purpose, we use the `omp task` directive in OpenMP API. Here we should consider data dependency among sub tasks, which is more strict in "aliased" cases than in "non-aliased" cases.

Figure 6 shows the pseudo code. Lines 9-22 show recursive calls in "aliased" cases. When all of A, B, C are the same sub matrix, the

```
1:  procedure FW-PARALLEL-REC($A, B, C$)
2:      if $A, B, C$ is smaller than a threshold then
3:          FW-BASE($A, B, C$)
4:      else
5:          Divide $A$ into $A_{00}, A_{01}, A_{10}, A_{11}$
6:          Divide $B$ into $B_{00}, B_{01}, B_{10}, B_{11}$
7:          Divide $C$ into $C_{00}, C_{01}, C_{10}, C_{11}$
8:          if location of $C$ is same as that of $A$ or $B$ then
9:              FW-PARALLEL-REC($A_{00}, B_{00}, C_{00}$)
10:             #pragma omp task
11:                 FW-PARALLEL-REC($A_{00}, B_{01}, C_{01}$)
12:             #pragma omp task
13:                 FW-PARALLEL-REC($A_{10}, B_{00}, C_{10}$)
14:             #pragma omp taskwait
15:             FW-PARALLEL-REC($A_{10}, B_{01}, C_{11}$)
16:             FW-PARALLEL-REC($A_{11}, B_{11}, C_{11}$)
17:             #pragma omp task
18:                 FW-PARALLEL-REC($A_{11}, B_{10}, C_{10}$)
19:             #pragma omp task
20:                 FW-PARALLEL-REC($A_{01}, B_{11}, C_{01}$)
21:             #pragma omp taskwait
22:             FW-PARALLEL-REC($A_{01}, B_{10}, C_{00}$)
23:         else
24:             #pragma omp task
25:                 FW-PARALLEL-REC($A_{00}, B_{00}, C_{00}$)
26:             #pragma omp task
27:                 FW-PARALLEL-REC($A_{00}, B_{01}, C_{01}$)
28:             #pragma omp task
29:                 FW-PARALLEL-REC($A_{10}, B_{00}, C_{10}$)
30:             #pragma omp task
31:                 FW-PARALLEL-REC($A_{10}, B_{01}, C_{11}$)
32:             #pragma omp taskwait
33:             #pragma omp task
34:                 FW-PARALLEL-REC($A_{11}, B_{11}, C_{11}$)
35:             #pragma omp task
36:                 FW-PARALLEL-REC($A_{11}, B_{10}, C_{10}$)
37:             #pragma omp task
38:                 FW-PARALLEL-REC($A_{01}, B_{11}, C_{01}$)
39:             #pragma omp task
40:                 FW-PARALLEL-REC($A_{01}, B_{10}, C_{00}$)
41:             #pragma omp taskwait
```

Figure 6: Parallel recursive divide-and-conquer FW algorithm. with parallel task creation.

algorithm has most severe dependencies[2]. First we compute C_{00}, aliased with A_{00} and B_{00} (line 9). By using the results, we can compute C_{01} and C_{02}. These two computation can be done in parallel (lines 10-13). Then we have to wait until two tasks with `taskwait` directive (line 14) and then we compute C_{11} (line 15). In latter half, we do the similar computation from C_{11} (lines 16-22).

[2]If we consider cases where only two matrices are aliased, we could alleviate the dependency. The current implementation does not distinguish such cases since its effects on performance are minor

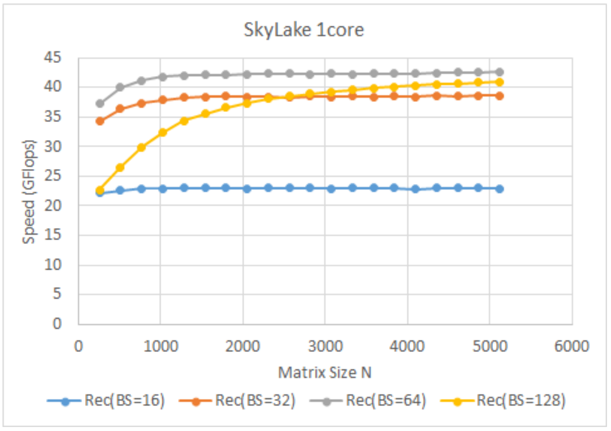

Figure 7: Performance with different block sizes on SkyLake 1core.

The parallel algorithm in "non-aliased" cases (lines 24-41) is simpler; 4 computations in the first half are done in parallel, and then computations in the latter half are done in parallel.

In recursive task creation, it is widely known that it has an advantage to stop task creation when the sub problem size gets smaller than a dedicated threshold, since task creation costs are suppressed. In our case with the compiler described in Table 1, however, we observed the highest performance if we create tasks until the sub problem size reaches *BS*, when the recursive call stops.

4 PERFORMANCE EVALUATION

4.1 Evaluation Conditions

This section describes performance evaluation of our FW implementation using SIMD instructions on a SkyLake machine and a KNL machine described in Table 1. On a KNL machine, we use "Flat" memory mode, in which DDR4 memory and MCDRAM memory are visible as two NUMA nodes. We show performance for when only DDR4 is used and MCDRAM is used. For this purpose, `numactl` Linux command is used to configure memory allocation policy.

4.2 Single-core Performance

First, we evaluate performance of our recursive implementation with several block sizes (BS) on a single core of the SkyLake machine (Figure 7). The x-axis of the graph corresponds to the problem size N and its y-axis is the performance in GFlops. Here we observe that $BS = 64$ exhibits higher performance than $BS = 16$ or $BS = 32$, while $BS = 128$ suffers slow down. In the best case of $BS = 64$, 42.6GFlops speed is achieved, which is 29% of peak performance of a single core, 147.2GFlops. As described in Section 2.2, for FW algorithm which does not use FMADD operations, the theoretical upper limit is halved and 73.6GFlops. Based on this value, the performance ratio is 58%[3]. Hereafter, we let $BS = 64$.

[3]Considering AVX-512 clock lower than base clock, the ratio is even higher

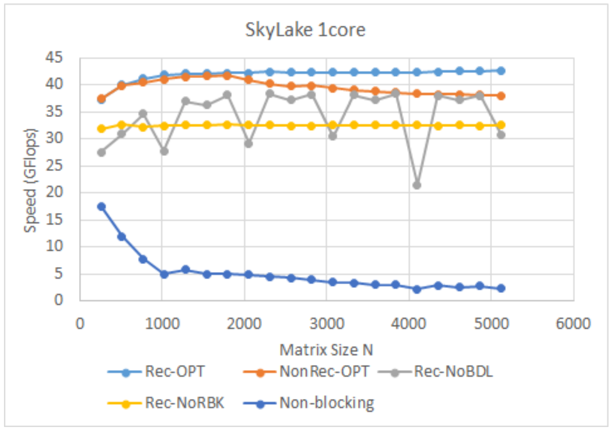

Figure 8: Effects of optimization techniques on performance on SkyLake 1core.

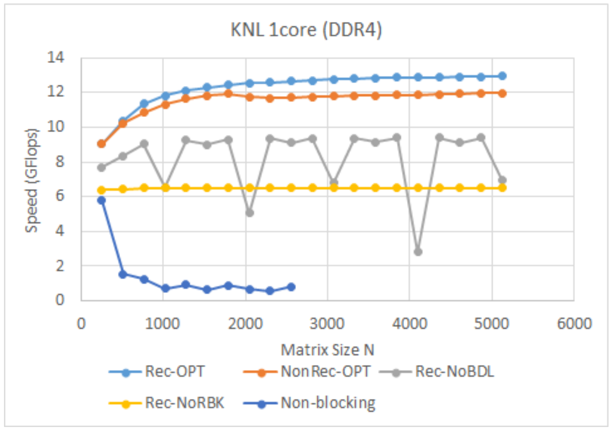

Figure 9: Effects of optimization techniques on performance on KNL 1core. DDR4 memory is used.

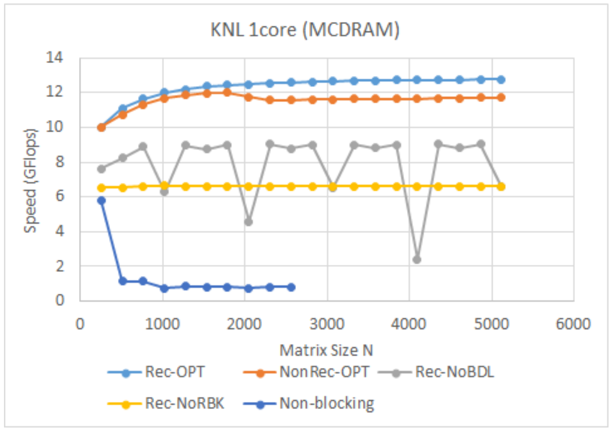

Figure 10: Effects of optimization techniques on performance on KNL 1core. MCDRAM memory is used.

Next, we evaluate effects of optimization techniques described in Section 3.1 on performance . The followings are compared on each machine configuration:

Rec-OPT: Recursive algorithm with optimization techniques.

NonRec-OPT: Blocked, non-recursive algorithm. Optimization techniques, block data layout (Section 3.1.2) and register blocking kernel (Section 3.1.3) are used.

Rec-NoBDL: Recursive algorithm. Block data layout (BDL) optimization is omitted.

Rec-NoRBK: Recursive algorithm. Usage of register blocking kernel (RBK) in non-aliased cases is omitted; a SIMD kernel in Figure 4 is always used.

NoBlocking: No blocking technique is used; this evaluation is done by applying the SIMD kernel to the entire input matrix.

Figures 8, 9, 10 show results on the SkyLake machine, the KNL machine (MCDRAM), the KNL machine (DDR4) machine, respectively. We observe that without blocking (*NoBlocking*), the performance is tremendously worse in all cases, although it works reasonably with very small matrices. The performance curves are different between SkyLake and KNL. While there is heavy slowdown $N = 512$ on KNL, the performance decrease is gradual on SkyLake. We consider this is due to absence of L3 cache on KNL; When $N = 512$, the matrix size reaches 1MiB, which is equal to L2 cache size on KNL. On SkyLake, we consider that larger L3 cache works to alleviate the slowdown.

The optimized recursive algorithm (*Rec-OPT*) achieves the best performance and stable against varying N. This demonstrates that recursive divide-and-conquer approach makes algorithms oblivious to cache/memory hierarchy. It achieves 42.6GFlops on SkyLake, 12.7GFlops on KNL with MCDRAM and 12.9GFlops on KNL with DDR4. On KNL, the performance ratio is around 15% (when half of peak is based, it is around 30%), while it is 29% on SkyLake. The reasons of this difference between SkyLake and KNL shall be analyzed in the near future.

The non-recursive blocked algorithm (*NonRec-OPT*) also works fairly well, and shows similar high performance when $N < 2048$. However, it suffers from slowdown up to 11% with larger N. We consider this reason as follows: by blocking with size BS, the algorithm can use L1 cache efficiently. On the other hand, when the size of a sub-matrix $N \times BS$ (0.5MiB with $N = 2048, BS = 64$) gets close to or larger than L2/L3 cache, non-recursive algorithm has a disadvantage compare with the recursive version.

Next, we evaluate the effect of the block data layout (BDL) optimization. The performance of *Rec-NoBDL* is lower than *Rec-OPT*. More notable feature is that it suffers heavy slow down when N is a multiple of 1024. When $N = 4096$, *Rec-NoBDL* is 50% and 80% slower than *Rec-OPT* on SkyLake and KNL, respectively. As described in Section 3.1.2, it is natural to consider this is due to conflict cache misses. The transformation of layout is necessary to make the performance stable against the change of problem sizes.

The performance of *Rec-NoRBK* explains the impact of using register blocking kernel (RBK). The performance is stable against N, however, it is significantly slower than *Rec-OPT*. Basically, the

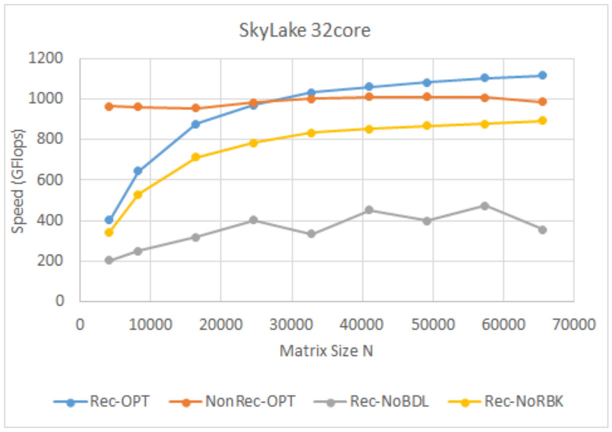

Figure 11: Effects of optimization techniques on parallel performance on SkyLake. 32 cores are used.

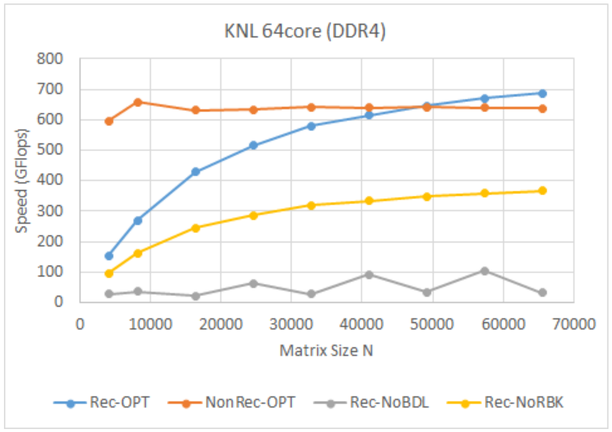

Figure 12: Effects of optimization techniques on parallel performance on KNL. 64 cores and DDR4 memory is used.

performance ratio of the two gets closer to ratio of the two kernels in Figure 4 and Figure 5 with sufficiently large N, since "non-aliased" cases are the majority. The difference is 24% on SkyLake and 48% on KNL.

4.3 Multi-core Performance

This section evaluates multi-core algorithm described in Section 3.2. Figures 11, 12, 13 shows performance with 32 cores on the SkyLake machine, that with 16 cores on the KNL machine with DDR4 memory, and that on the same condition with MCDRAM memory, respectively.

On SkyLake, *Rec-Opt* achieves 1117GFlops at $N = 65536$ with 32 cores, which is 28.2 times higher than 1-core case. On KNL, the performance is 686-687GFlops with 64 cores, 53-54 times higher than 1-core case. Unlike in the single-core case, the performance largely depends on N.

We observe performance of *NonRec-Opt* is more stable against N. We consider the difference in behavior between the two comes

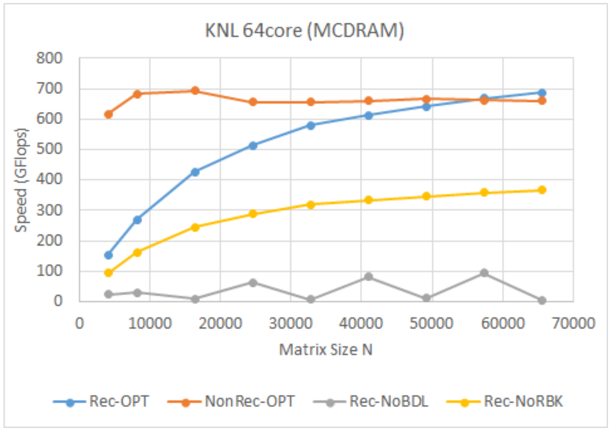

Figure 13: Effects of optimization techniques on parallel performance on KNL. 64 cores and MCDRAM memory is used.

from difference in costs of "`omp task`" directive frequently used in *Rec-Opt* and "`omp for`" in *NonRec-Opt*. Performance of *Rec-Opt* gets higher with larger N, and it gets over *NonRec-Opt* when $N \geq 32768$ on SkyLake. We see similar tendency on KNL, however, the cross points are at different places. In the discussion so far, there were not notable differences between MCDRAM and DDR4 on KNL. Here we observe a difference in the performance of *NonRec-Out*; it stays around 640GFlops with DDR4 and 660GFlops with MCDRAM, which brings about different cross points between the two. We consider this is caused by that MCDRAM has higher bandwidth than DDR4.

We observe that the speed down in *Rec-NoBDL* is heavier than in single-core case. We suppose this is due to costs for conflict cache misses that occur in parallel, which increase demands for main memory bandwidth. We observe differences here between MCDRAM and DDR4 again; with DDR4, speeds of *Rec-NoBDL* are 20 to 30 GFlops when N is a multiple of 16384. With MCDRAM, on the other hand, they are only around 10 GFlops. This contradicts with the fact that MCDRAM has higher bandwidth; we will investigate this point in future.

Rec-NoRBK shows a similar tendency to the single-core case. On KNL, its speed is 365 GFlops at $N = 65536$, 47% lower than *Rec-Opt*. This number is close to 338GFlops reported by Rucci et al.[11]. This result indicates that the register blocking technique is a key for high performance, even in the blocked algorithm with sufficiently low cache miss ratio.

5 CONCLUSION

This paper described a parallel high performance implementation of Floyd-Warshall (FW) algorithm. This is designed to harness properties of modern processors, deep cache hierarchy, SIMD parallelism and multi-core parallelism. Its basic strategy is to combine optimized kernels for AVX-512 and recursive divide-and-conquer approach for cache-obliviousness. In addition to that, we have shown that several techniques that are common in dense linear algebra software are keys to improve performance. By integrating block data format and register blocking, the implementation achieved 1.1TFlops on a dual-socket Skylake machine and 700GFlops a KNL machine.

We have reported a performance issue on multi-core version when the problem size is not very large. In these cases, it is slower than non-recursive method with a traditional blocking. To improve this, we will need task parallel runtimes with higher performance and scalability.

Another issue is related to productivity of software. In spite of the basic idea of cache-oblivious approach, which minimizes architecture dependent parameters, our current implementation includes architecture dependent part. When we port this software to architecture with different SIMD instruction sets, such as AVX-2, ARM Neon and ARM scalable vector extension (SVE), we need to rewrite kernels and optimize them. More portable way to generate kernels (semi) automatically is required. Also it will be interesting to utilize SVE, which supports different processor generations with different vector length without changing binary code.

By harnessing knowledge and experiments through implementation of cache-oblivious FW algorithm, we will apply the techniques to other applications, including graph problems with more irregular structures, and kernels of deep learning and machine learning.

Acknowledgements This research is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO) and RWBC-OIL, AIST, Japan.

REFERENCES

[1] Robert Lucas et. al.: Top Ten Exascale Research Challenges, DOE ASCAC Subcommittee Report (2014).
[2] Kazushige Goto and Robert van de Geijn: Anatomy of high-performance matrix multiplication, ACM Trans. Math. Softw., vol. 34, no. 3, pp. 1–25 (2008).
[3] R. Clint Whaley, Antoine Petitet, Jack Dongarra, Automated empirical optimizations of software and the ATLAS project, Parallel Computing, Volume 27, Issues 1–2, pp. 3-35 (2001).
[4] Jack Dongarra, Piotr Luszczek, Antoine Petitet, The LINPACK Benchmark: past, present and future, Concurrency and Computation: Practice and Experience, Volume 15, Issue 9, pp. 803-820 (2003).
[5] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of sparse matrix-vector multiplication on emerging multicore platforms, SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, pp. 1-12 (2007).
[6] Jee W. Choi, Amik Singh, Richard W. Vuduc, Model-driven Autotuning of Sparse Matrix-Vector Multiply on GPUs, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 115-126 (2010).
[7] M. E. Wolf and M. S. Lam: A Data Locality Optimizing Algorithm. Proceedings of ACM PLDI 91, pp. 30–44 (1991).
[8] M. Wittmann, G. Hager, and G. Wellein: Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory. Workshop on Large-Scale Parallel Processing (LSPP10), in conjunction with IEEE IPDPS2010, 7pages (2010).
[9] Takeshi Fukaya,Takeshi Iwashita: Time-space tiling with tile-level parallelism for the 3D FDTD method. Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2018), pp. 116-126 (2018).
[10] Benjamin Diament and Andras Ferencz, Comparison of Parallel APSP Algorithms (1999).
[11] Enzo Rucci, Armando De Giusti, and Marcelo Naiouf: Blocked All-Pairs Shortest Paths Algorithm on Intel Xeon Phi KNL Processor: A Case Study, Proceedings of CACIC 2017 : XXIII Argentine Congress of Computer Science (CACIC 2017), pp. 47-57 (2017).
[12] James Jeffers, James Reinders, Avinash Sodani: Intel Xeon Phi Processor High Performance Programming 2nd Edition, Knights Landing Edition, Morgan Kaufmann (2016).
[13] Matteo Frigo, Charles Leiserson, Harald Prokop, Sridhar Ramachandran: Cache-oblivious algorithms, Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pp. 285–297 (1999).
[14] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. : A locality preserving cache-oblivious dynamic dictionary. Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 29–38 (2002).
[15] Matteo Frigo and Volker Strumpen: Cache Oblivious Stencil Computations. In Poceedings of ACM International Conference on Supercomputing (ICS'05), pp. 361-366 (2005).
[16] Toshio Endo: Applying Recursive Temporal Blocking for Stencil Computations to Deeper Memory Hierarchy, Proceedings of the 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA 2018), pp. 19-24 (2018).
[17] Joon-Sang Park, Michael Penner, and Viktor K Prasanna: Optimizing Graph Algorithms for Improved Cache Performance, IEEE Transactions on parallel and distributed systems, vol. 15, issue 9, pp. 769-782 (2004).
[18] P. D'Alberto and A. Nicolau. R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks, Algorithmica, 47(2):203–213 (2007).
[19] Aydin Buluc, John R. Gilbert, Ceren Budak: Gaussian Elimination Based Algorithms on the GPU, UCSB Technical Report, 2008-15 (2008).
[20] Intel: Intrinsics Guide, `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`.
[21] Intel: Intel Xeon Processor Scalable Family, `https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf` (2019).
[22] F. G. Gustavson: New generalized data structures for matrices lead to a variety of high performance algorithms. Proceedings of the International Conference on Parallel Processing and Applied Mathematics (PPAM), pp. 418–436, Springer-Verlag (2002).
[23] Neungsoo Park, Bo Hong, Viktor K. Prasanna: Tiling, Block Data Layout, and Memory Hierarchy Performance, IEEE Transactions on Parallel and Distributed Systems, Vol 14, No 7, pp. 640-654 (2003).
[24] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, Jim Demmel: Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology, Proceedings of the 11th international conference on Supercomputing (ICS 97), pp. 340-347 (1997).
[25] R. Vuduc, J.W. Demmel, K.A. Yelick, S. Kamil, R. Nishtala, B. Lee: Performance Optimizations and Bounds for Sparse Matrix-Vector Multiply, Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC02) (2002),