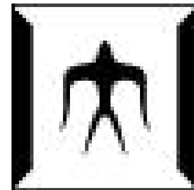
Integrating Cache Oblivious Approach with Modern Processor Architecture: *The Case of Floyd-Warshall Algorithm*

Toshio Endo (遠藤敏夫)
GSIC, Tokyo Institute of Technology (東京工業大学)

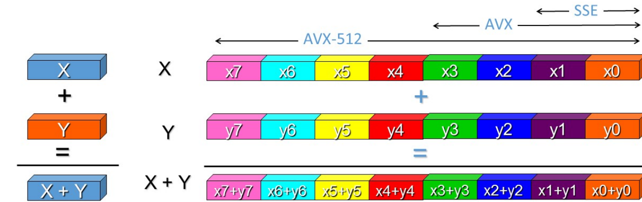


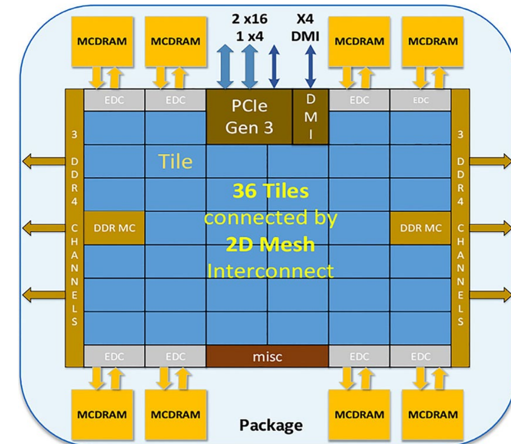
Supported by NEDO and RWBC-OIL, AIST

Architecture Trends

- Each processor has more and more cores
  - Recent Xeon/EPYC have up to 56/64 cores
- Each core gains higher FLOPs with SIMD instructions
  - AVX-2, AVX-512, SVE…
- In order to mitigate memory-wall problem, modern architecture tends to have
  - Deeper cache hierarchy
    - L1 ⇔ L2 ⇔ L3 ⇔ Main memory
  - Hybrid memory including High-bandwidth memory or NVM

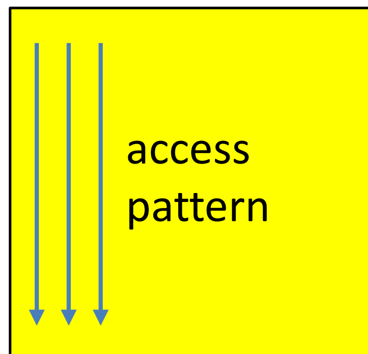
➔ Algorithm kernels has been & need to be reconsidered

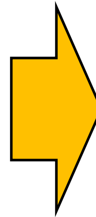


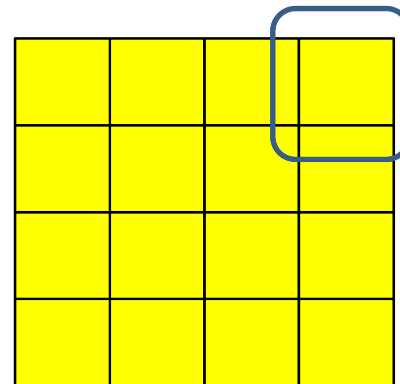


Cache Blocking

- Cache blocking is one of standard techniques to improve locality
- Used to accelerate
  - Dense/sparse linear algebra
  - Stencil computation
  - Graph algorithms, etc.



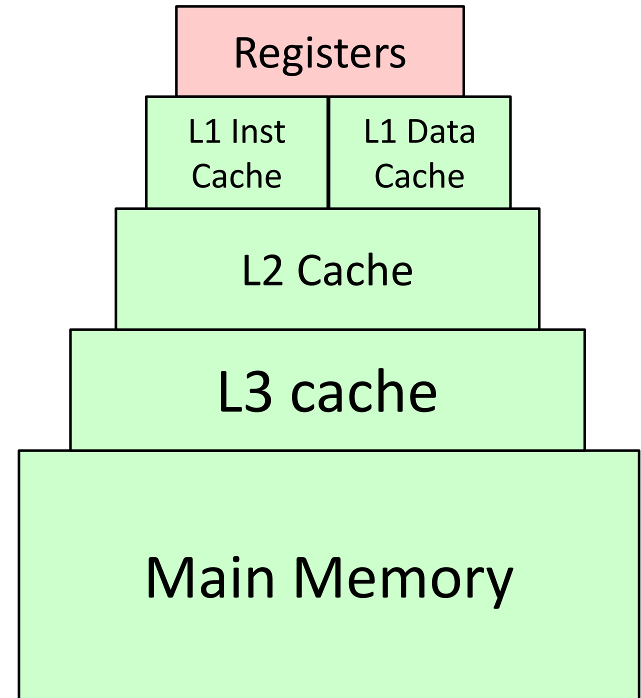




Block size
< Cache size

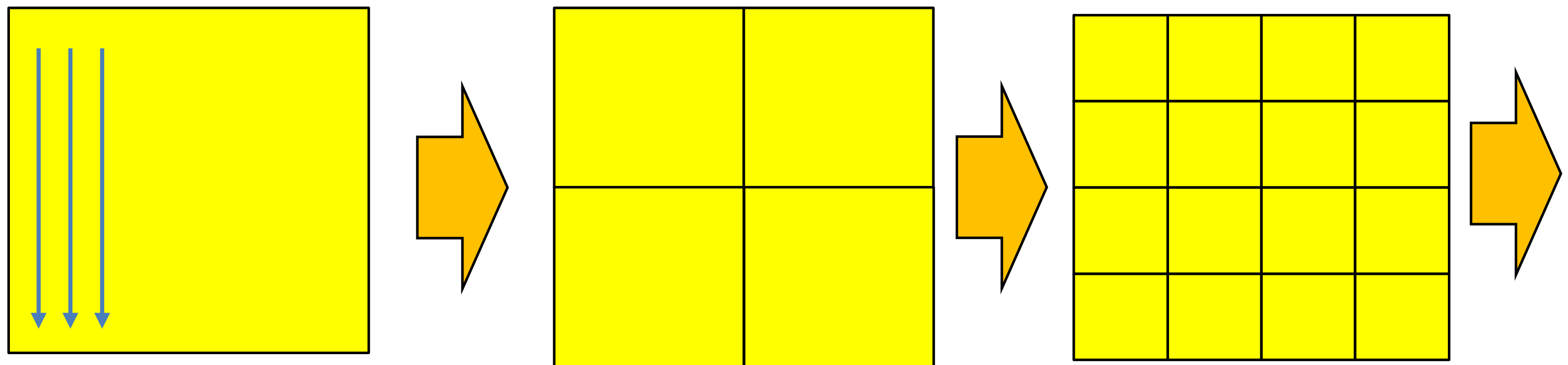
Issues of Cache Blocking

- Block sizes need to be architecture aware
  - Sizes of each cache level
  - Number of cache levels
    - cf: Typical HPC CPUs have 3 level, while Xeon Phi have 2 level
- If we support multi-level blocking, programming gets harder



Cache-Oblivious Approach

- Cache-oblivious approach has been proposed [Frigo et al. 99]
- Recursive divide & conquer is used to make the working set size fit size of each cache level

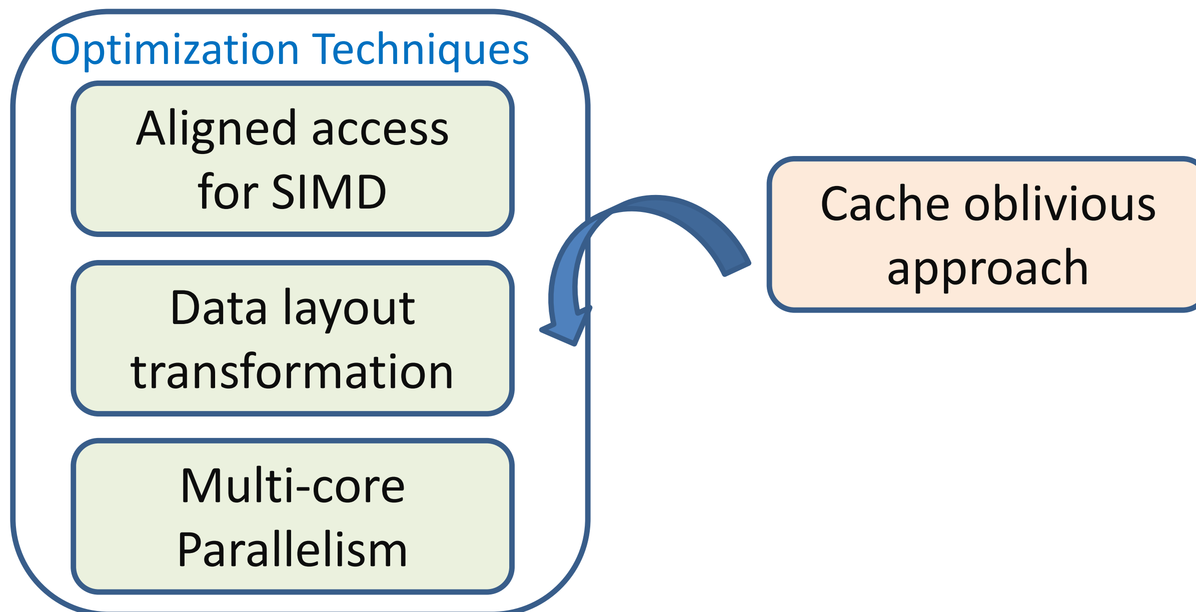


➔ This approach makes algorithms more architecture independent

– Applied to linear algebra kernel, stencil, graph, FFT…

Locality is a Big Issue, But We Have More

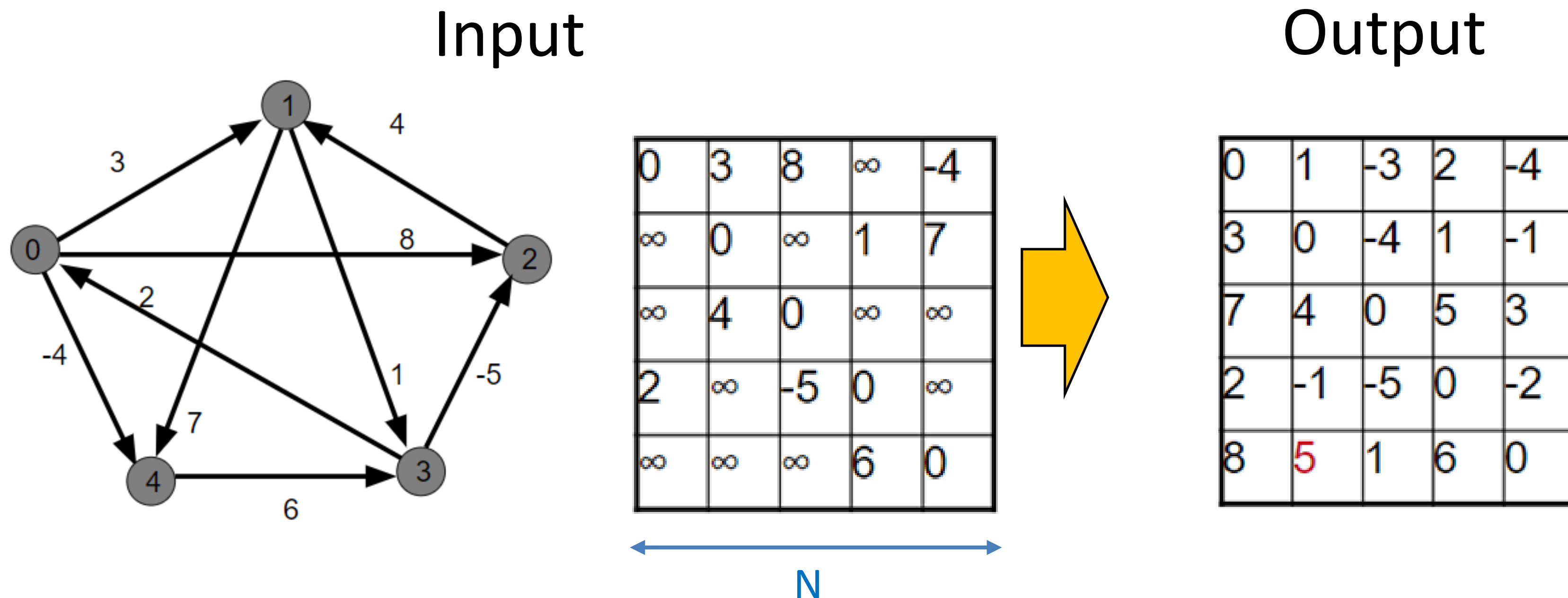
- Cache oblivious approach improve locality for multiple level of caches
- However, we need to investigate whether it works well with considerations of other features in modern processors
  - SIMD parallelism
  - Multi/many core parallelism



Our Target Algorithm: Floyd-Warshall Algorithm

Floyd-Warshall (FW)algorithm:

- A well-known algorithm for All-pairs Shortest Path (APSP) problem in graph analysis



0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

Summary of This Work

- A high performance FW implementation is given
  - Works with AVX-512 SIMD instructions
  - Supports multi-core
  - Based on cache-oblivious approach
- 1.1 TFlops on dual Skylake Xeon
- 700 Gflops on Xeon Phi KNL
  - In single precision
- https://github.com/toshioendo/hoalgos

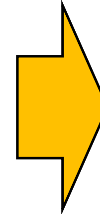
Non-Blocked FW Algorithm

D: a distance matrix of size N

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

N

D[i,j]: the weight of the edge from i to j



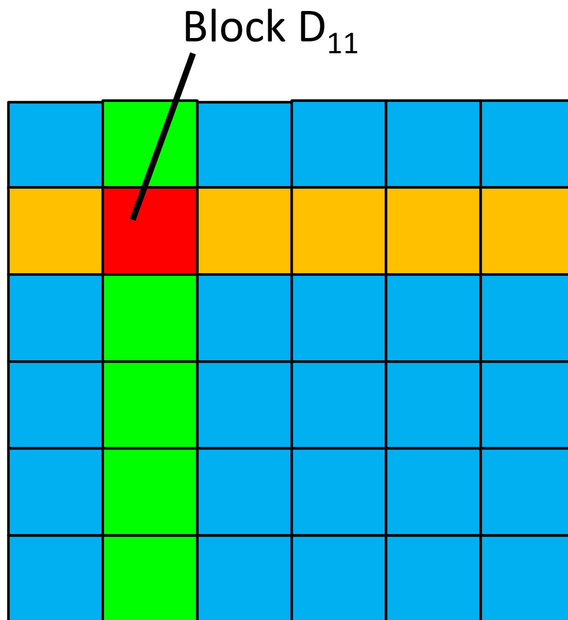
0	1	-3	2	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

D[i,j]: the length of shortest path from i to j

procedure FW(D)
    for $k = 0, \ldots, N-1$ **do**
        for $i = 0, \ldots, N-1$ **do**
            for $j = 0, \ldots, N-1$ **do**
                if $D[i,j] > D[i,k] + D[k,j]$ **then**
                    $D[i,j] = D[i,k] + D[k,j]$

Complexity: $O(N^3)$

(Non-Recursive) Blocked FW Algorithm



BS

Main algorithm

procedure FW-Blocking(D)

for k = 0 … N/BS-1

FW-BASE(D_{kk}, D_{kk}, D_{kk})

for all D_{kj}

FW-BASE(D_{kk}, D_{kj}, D_{kj})

for all D_{ik}

FW-BASE(D_{ik}, D_{kk}, D_{ik})

for all D_{ij}

FW-BASE(D_{ik}, D_{kj}, D_{kj})

Base kernel (Block-wise)

procedure FW-BASE(A, B, C)
 for $k = 0, \ldots, BS - 1$ **do**
  for $i = 0, \ldots, BS - 1$ **do**
   for $j = 0, \ldots, BS - 1$ **do**
    if $C[i, j] > A[i, k] + B[k, j]$ **then**
     $C[i, j] = A[i, k] + B[k, j]$

Recursive Blocking FW Algorithm

[Park et al. 04]

1st half

B_{00}	B_{01}
B_{10}	B_{11}

A_{00}	A_{01}
A_{10}	A_{11}

C_{00}	C_{01}
C_{10}	C_{11}

2nd half

B_{00}	B_{01}
B_{10}	B_{11}

A_{00}	A_{01}
A_{10}	A_{11}

C_{00}	C_{01}
C_{10}	C_{11}

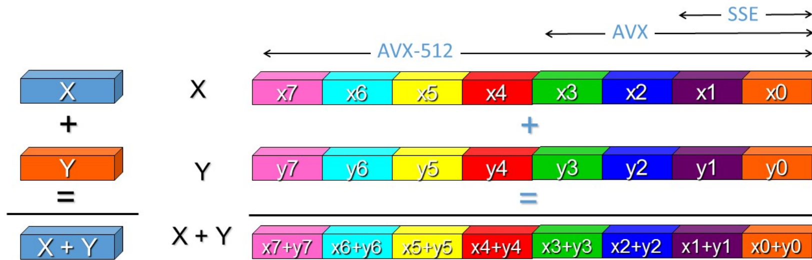
procedure FW-REC(A, B, C)
  if A, B, C is smaller than a threshold (=BS) **then**
    FW-BASE(A, B, C) → Stop recursion
  else
    Divide A into $A_{00}, A_{01}, A_{10}, A_{11}$
    Divide B into $B_{00}, B_{01}, B_{10}, B_{11}$
    Divide C into $C_{00}, C_{01}, C_{10}, C_{11}$
    FW-REC(A_{00}, B_{00}, C_{00})
    FW-REC(A_{00}, B_{01}, C_{01})
    FW-REC(A_{10}, B_{00}, C_{10})
    FW-REC(A_{10}, B_{01}, C_{11})
    FW-REC(A_{11}, B_{11}, C_{11})
    FW-REC(A_{11}, B_{10}, C_{10})
    FW-REC(A_{01}, B_{11}, C_{01})
    FW-REC(A_{01}, B_{10}, C_{00})

procedure FW(D)
  FW-REC(D, D, D)

Integration with Optimizations for Modern Processors

- So far, cache oblivious approach has been adopted
- Furthermore, we need to introduce optimizations for modern processors
  - SIMD parallelism
  - Data layout transformation
  - Multi-core parallelism
  - Kernel optimization with register blocking

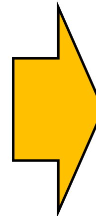
Acceleration with AVX-512 SIMD Instructions [Rucci et al. 17]



With AVX-512, 16 SP values are computed at once

procedure FW-BASE(A, B, C)
  for $k = 0, \ldots, BS - 1$ **do**
    for $i = 0, \ldots, BS - 1$ **do**
      for $j = 0, \ldots, BS - 1$ **do**
        if $C[i, j] > A[i, k] + B[k, j]$ **then**
          $C[i, j] = A[i, k] + B[k, j]$

"c = *min (c, a+b)*"



BS should be a multiple of 16

procedure FW-BASE-SIMD(A, B, C)
  for $k = 0, \ldots, BS - 1$ **do**
    for $i = 0, \ldots, BS - 1$ by 16 **do**
      a = _mm512_loadu_ps(&$A[i, k]$)
      for $j = 0, \ldots, BS - 1$ **do**
        b = _mm512_set1_ps($B[k, j]$)
        c = _mm512_loadu_ps(&$C[i, j]$)
        sum = _mm512_add_ps(a, b)
        $mask$ = _mm512_cmp_ps_mask
          (sum, c, _CMP_LT_OQ)
        _mm512_mask_storeu_ps(&$C[i, j], mask, sum$)

"*min (c, a+b)*" is achieved by using a mask register

Introducing Block Data Layout

- With cache blocking, memory access pattern is improved
- However, we may still suffer from conflict cache misses with the standard column major format

➔ Block data layout is adopted

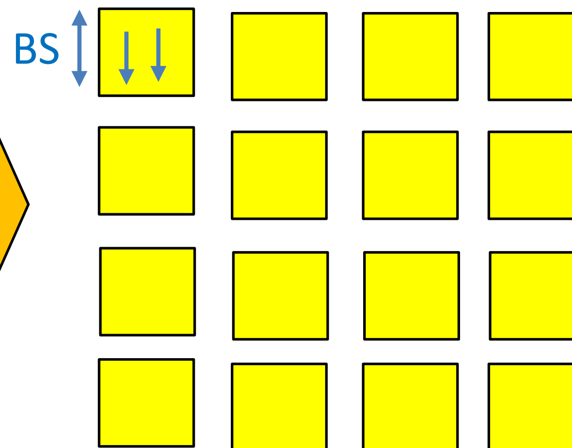
➔ Layout transformation is done before&after FW computation

(performance measurement includes this overhead)

column major format

consecutive memory address

block data layout



Note: recursive division sizes should be a multiple of BS

Acceleration with Muti-Core Parallelism

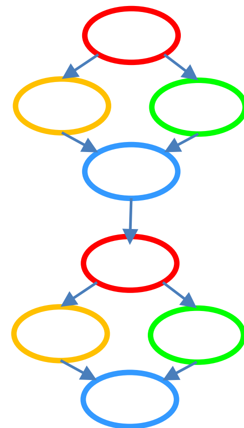
- OpenMP is used

Non-recursive blocked algorithm

procedure FW-Blocking(D)
for k = 0 … N/BS-1
FW-BASE(D_{kk}, D_{kk}, D_{kk})
omp for → *for all D_{kj}*
FW-BASE(D_{kk}, D_{kj}, D_{kj})
omp for → *for all D_{ik}*
FW-BASE(D_{ik}, D_{kk}, D_{ik})
omp for → *for all D_{ij}*
FW-BASE(D_{ik}, D_{kj}, D_{kj})

Recursive blocked algorithm

procedure FW-REC(A, B, C)
if A, B, C is smaller than a threshold **then**
FW-BASE(A, B, C)
else
Divide A into $A_{00}, A_{01}, A_{10}, A_{11}$
Divide B into $B_{00}, B_{01}, B_{10}, B_{11}$
Divide C into $C_{00}, C_{01}, C_{10}, C_{11}$
omp task → FW-REC(A_{00}, B_{00}, C_{00})
omp task → FW-REC(A_{00}, B_{01}, C_{01})
omp taskwait → FW-REC(A_{10}, B_{00}, C_{10})
FW-REC(A_{10}, B_{01}, C_{11})
omp task → FW-REC(A_{11}, B_{11}, C_{11})
omp task → FW-REC(A_{11}, B_{10}, C_{10})
omp taskwait → FW-REC(A_{01}, B_{11}, C_{01})
FW-REC(A_{01}, B_{10}, C_{00})



Re-visiting Base Kernel (1)

```
procedure FW-BASE-SIMD(A, B, C)
    for k = 0, . . . , BS − 1 do                       k loop is at outermost
        for i = 0, . . . , BS − 1 by 16 do
            a = _mm512_loadu_ps(&A[i, k])
            for j = 0, . . . , BS − 1 do
                b = _mm512_set1_ps(B[k, j])
                c = _mm512_loadu_ps(&C[i, j])          16 elements are read from C
                sum =_mm512_add_ps(a, b)
                mask =_mm512_cmp_ps_mask
                        (sum, c, _CMP_LT_OQ)
                _mm512_mask_storeu_ps(&C[i, j], mask, sum)   16 elements are written to C
```

➔ Every element in C is read from/written to memory **for BS times**

This "inefficiency" is required preserve data dependency

- Data written to C in k-th step may be read (as A or B) in k'-th step (k'>k)

➔ Loop interchange is illegal in such cases

Re-visiting Base Kernel (2)

procedure FW-BASE-SIMD(A, B, C) 🡐 Blocks A, B are read and C is written

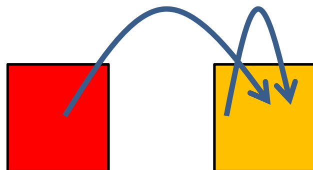
Do we always need to preserve dependency? ➔ No!

Aliased cases

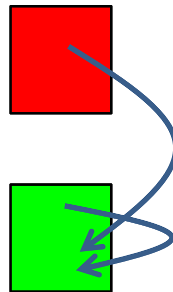
If (A=C and/or B=C), we have to preserve data dependency

(1) A=B=C

(2) A!=B=C



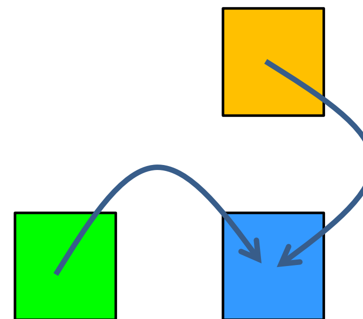
(3) B!=A=C



Non-aliased cases

If (A!=C and B!=C), we have opportunities for loop interchange optimization

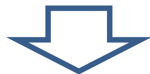
(4) A!=C && B!=C



Optimized Kernel with Loop Interchange and Register Blocking

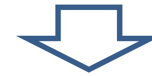
This kernel can be used only when A!=C and B!=C

procedure FW-Base-RegBlock(A, B, C)
  for $i = 0, \ldots, BS - 1$ by 16 **do**
    for $j = 0, \ldots, BS - 1$ by 16 **do**
      $c0$ = _mm512_set1_ps (∞)
      ...
      $c15$ = _mm512_set1_ps (∞)
      for $k = 0, \ldots, BS - 1$ **do**
        a = _mm512_loadu_ps (&$A[i, k]$)
        // for $c0$
        b = _mm512_set1_ps($B[k, j + 0]$)
        sum =_mm512_add_ps(a, b)
        $mask$ =_mm512_cmp_ps_mask
          ($sum, c0$, _CMP_LT_OQ)
        $c0$ = _mm512_mask_blend_ps($mask, c0, sum$)
        ...
        // for $c15$
        b = _mm512_set1_ps($B[k, j + 15]$)
        sum =_mm512_add_ps(a, b)
        $mask$ =_mm512_cmp_ps_mask
          ($sum, c15$, _CMP_LT_OQ)
        $c15$ = _mm512_mask_blend_ps($mask, c15, sum$)



16 accumulators are used

Now k loop is inner



// for $c0$
$tmpc$ = _mm512_loadu_ps(&$C[i, j + 0]$)
$mask$ =_mm512_cmp_ps_mask
  ($c0, tmpc$, _CMP_LT_OQ)
_mm512_mask_storeu_ps(&$C[i, j + 0], mask, c0$)
...
// for $c15$
$tmpc$ = _mm512_loadu_ps(&$C[i, j + 15]$)
$mask$ =_mm512_cmp_ps_mask
  ($c15, tmpc$, _CMP_LT_OQ)
_mm512_mask_storeu_ps(&$C[i, j + 15], mask, c15$)

After k loop finishes, memory read/write to C occur only once per element

Floyd-Warshall Implementations

	Park et al. 04	Rucci et al. 17	Ours
Cache Blocking	Yes	Yes	Yes
Recursive Cache Blocking	Yes	-	Yes
SIMD Parallelism	-	Yes	Yes
Block Data Layout	Yes	?	Yes
Multi-core Parallelism	-	Yes	Yes
Register Blocking	-	-	Yes

Experimental Environments

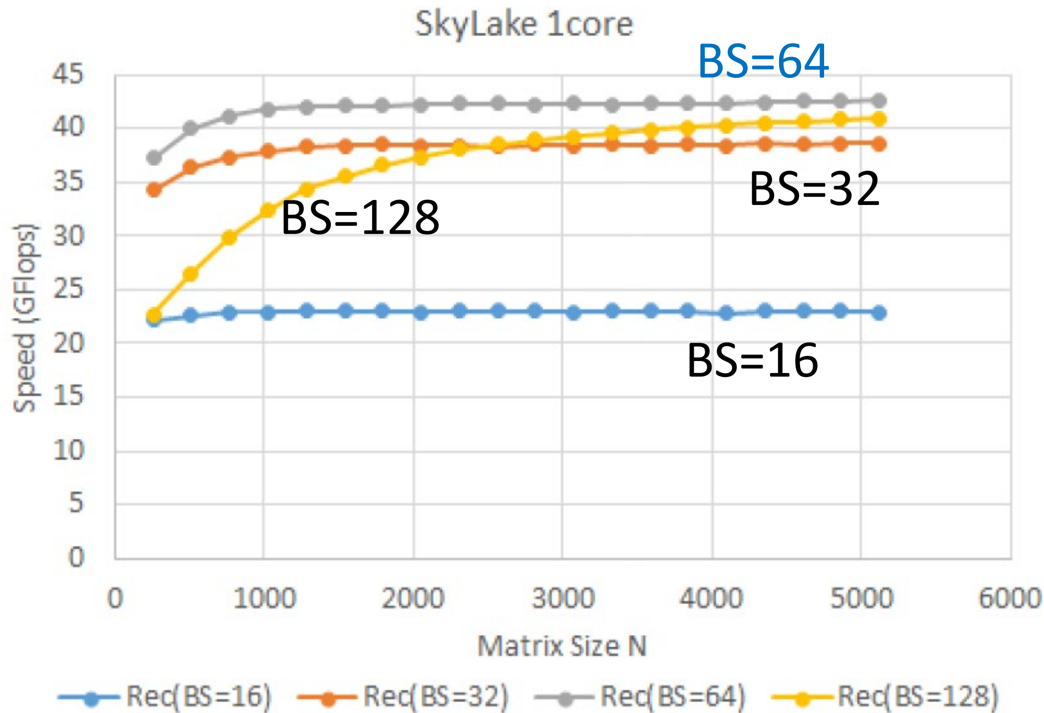
2 machines, both of which support AVX-512 are used

- 2-socket Xeon Skylake
- Xeon Phi KNL

	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

Block Size Configuration

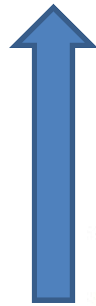
- Even with cache oblivious approach, we still have to determine a single parameter, block size (BS)



From the result of preliminary evaluation, we use BS=64

Performance Evaluation: 1-Core SkyLake

Faster



SkyLake 1core

Speed (GFlops)

45
40
35
30
25
20
15
10
5
0

0 1000 2000 3000 4000 5000 6000

Matrix Size N

Rec-OPT
NonRec-OPT
Rec-NoBDL
Rec-NoRBK
Non-blocking

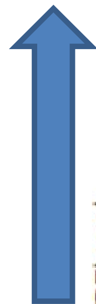
Non-Recursive is good, but Recursive achieves the fastest speed!!

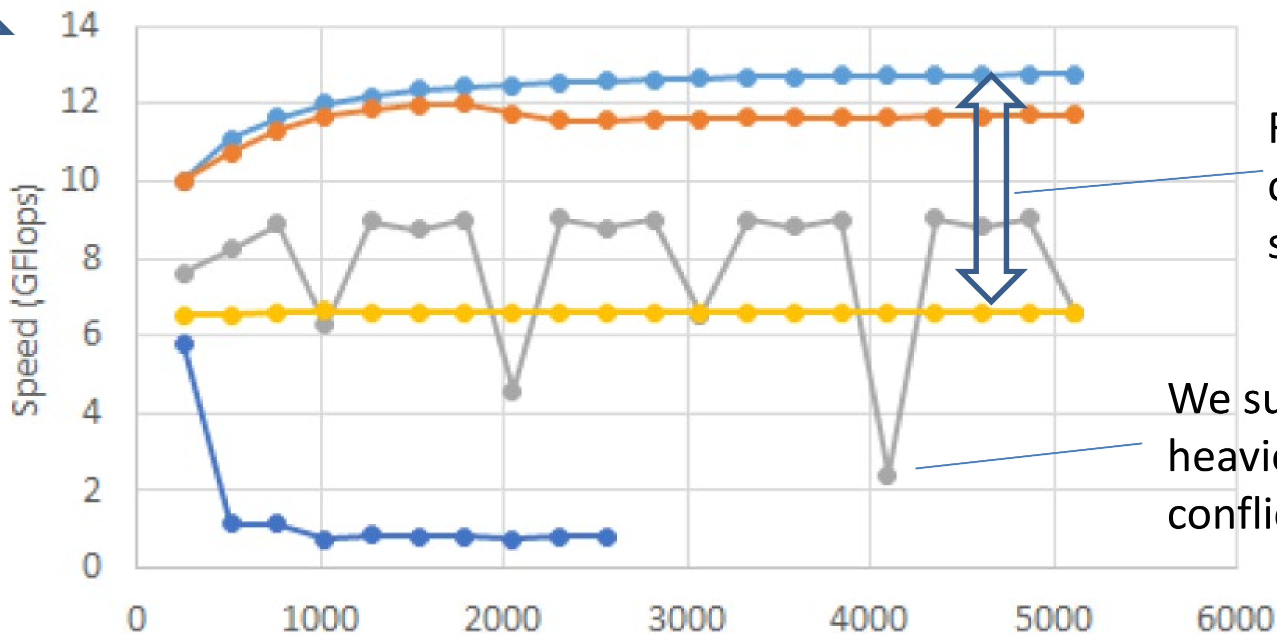
Register blocking contributes +30% speed-up

W/o layout change, we see slowdown when N is a multiple of 1024

Performance Evaluation: 1-Core KNL

Faster





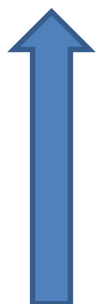
Register blocking contributes +100% speed-up

We suffer from heavier impacts of conflict misses!

Matrix D is put on MCDRAM; using DDR4 showed similar performance (refer to the paper)

Performance Evaluation: (16+16)-Core SkyLake

Faster



SkyLake 32core

cross point

Speed (GFlops)

1200
1000
800
600
400
200
0

0 10000 20000 30000 40000 50000 60000 70000

Matrix Size N

Rec-OPT NonRec-OPT Rec-NoBDL Rec-NoRBK

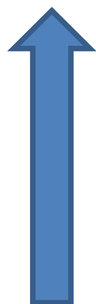
Recursive version exceeds 1.1TFlops!!

On the other hand, our recursive version gets slower with smaller N ☹

- Overhead of “omp task” ?

Performance Evaluation: 64-Core KNL

Faster



KNL 64core (MCDRAM)

Speed (GFlops): 0, 100, 200, 300, 400, 500, 600, 700, 800

Matrix Size N: 0, 10000, 20000, 30000, 40000, 50000, 60000, 70000

Rec-OPT, NonRec-OPT, Rec-NoBDL, Rec-NoRBK

cross point

~0.7TFlops !!

Conflict misses make execution completely impractical

We see that slow-down with smaller N is heavier

Peak Performance Ratio

- 2-socket Skylake:
  - Measured: 1.117 TFlops
  - Peak (SP): 5.304 TFlops

  ➔ Peak perf ratio=21%

  If we do not count FMAD in peak, ratio=42%
- KNL:
  - Measured: 0.687 TFlops
  - Peak (SP): 5.324 TFlops

  ➔ Peak perf ratio=13%

  If we do not count FMAD in peak, ratio=26%

NOTE: In FW, FMAD cannot be used efficiently

Summary

- A high performance FW implementation is given
  - Cache-oblivious approach is integrated with
    - SIMD parallelism
    - Multi-core parallelism
    - Data layout transformation
    - Register blocking with careful algorithm analysis
- Succeeds performance of state-of-art implementations
  - 1.1 TFlops on dual Skylake Xeon
  - 700 Gflops on Xeon Phi KNL
    - In this experiment, MCDRAM and DDR4 worked similarly

https://github.com/toshioendo/hoalgos

Future Work

- Evaluation of other heterogeneous memory
  - DIMM type 3D-Xpoint
- Towards further performance implementation
  - Reducing overhead of task creation by “omp task”
  - Improving memory affinity
    - Recursion + task creation works worse in this aspect

  ➔ Need improved multi-task runtime
- Towards more “architecture-independent” implementation
  - Our current version is free from cache-size parameter, but
  - The base kernel depends on SIMD-type and width

  ➔ ARM SVE (scalable vector extension) looks attractive