Performance Tuning for High Performance Computing Applications

Daisuke Takahashi

University of Tsukuba, Japan

Research Interests

- High-performance computing
- Developing parallel numerical libraries
  - Fast Fourier transform (FFT)
    - “FFTE library” http://www.ffte.jp/
    - FFTE's 1-D parallel FFT routine has been incorporated into the HPC Challenge (HPCC) benchmark.
  - Multiple-precision arithmetic
  - Linear algebra
- Performance tuning
  - Code optimization (parallelization, vectorization, etc.)
  - Memory optimization (cache blocking, etc.)

Outline

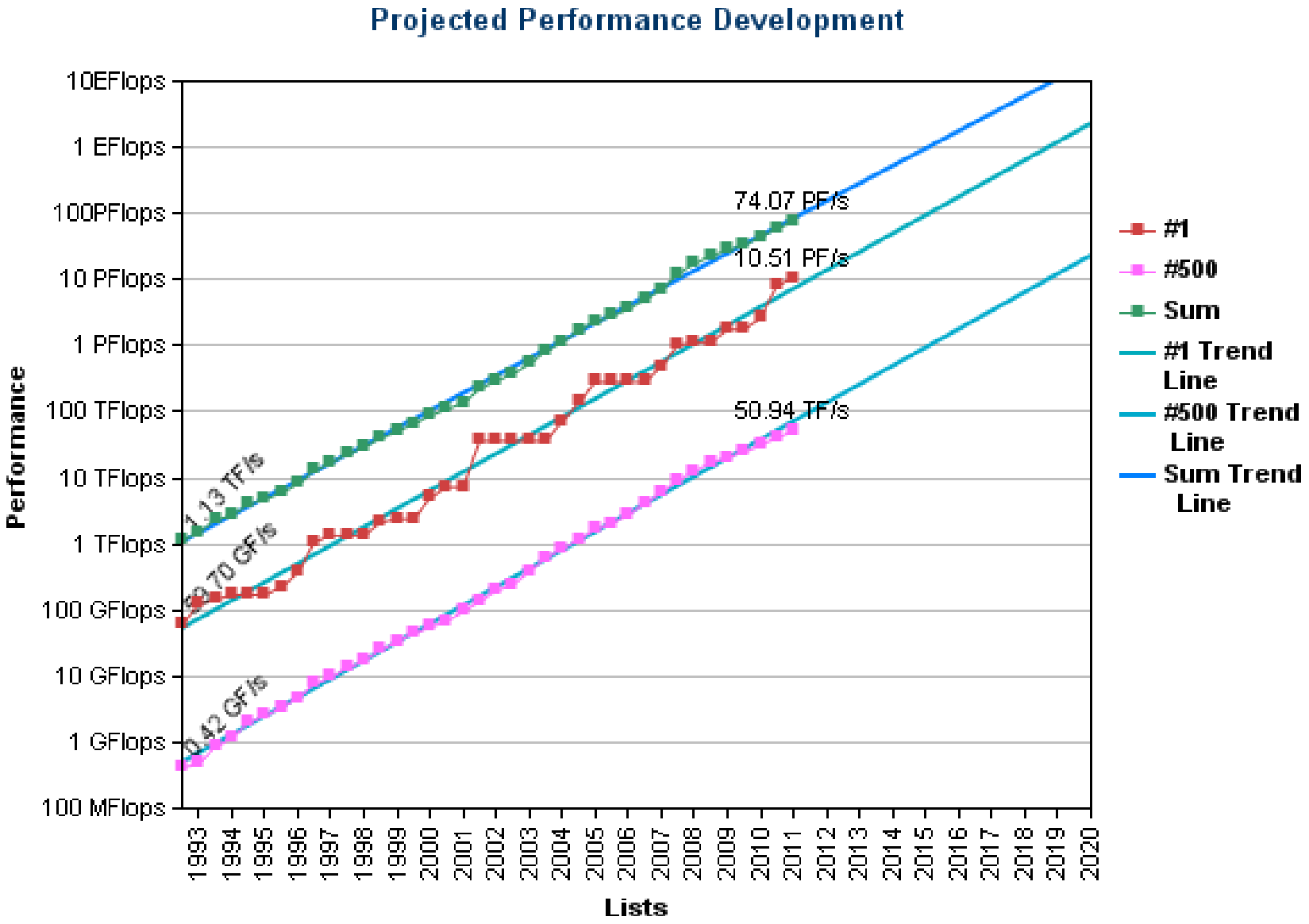
- Performance development of supercomputers
- It's all bandwidth
- Performance tuning
  - What is performance tuning?
  - Program optimization methods
- Implementation of parallel 1-D FFT using AVX instructions on multi-core processors

Performance Development of Supercomputers

- November 2011 TOP500 Supercomputing Sites
  - K computer (SPARC VIIIfx 8-core 2 GHz)
    10.51 PFlops (705,024 Cores)
  - Tianhe-1A (X5670 2.93 GHz 6-core, NVIDIA C2050)
    2.566 PFlops (186,368 Cores)
  - Jaguar (Cray XT5-HE 6-core 2.6 GHz)
    1.759 PFlops (224,162 Cores)
- Recently, the number of cores keeps increasing.

The K Computer

Source: http://www.top500.org/



Source: http://www.top500.org/

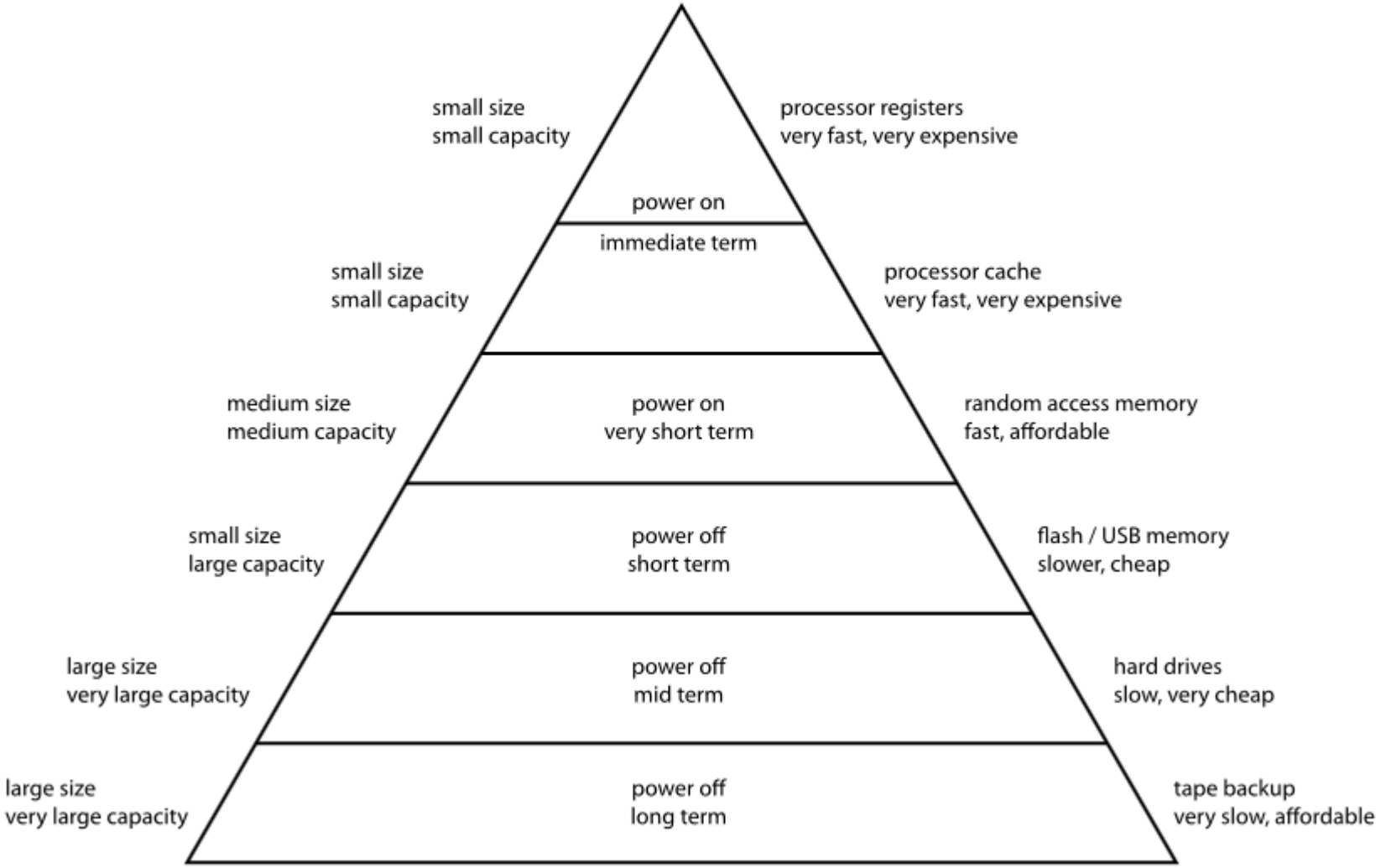
Linpack Benchmark

- Developed by Jack Dongarra of the University of Tennessee.
- Benchmark test for evaluating floating-point processing performance
- Uses Gaussian elimination method to estimate the time required for solving simultaneous linear equations
- Also used for the “TOP500 Supercomputer” benchmark

Indicator of Capability for Supplying Data to the Processor

- In a computer system that performs scientific computations, the “capability for supplying data to the processor” is most important.
- Unless data is supplied to the arithmetic unit of the processor, computations cannot be performed.
- The computing performance of the processor is largely impacted by the data supply capacity.
- “Bandwidth” is used as an indicator of the data supply capability.

Computer Memory Hierarchy



Source: Wikipedia

Memory Hierarchy (1/2)

- Memory hierarchy is designed based on the assumed locality of patterns of access to the memory area.
- Different types of locality:
  - Temporal locality
    - Property whereby the accessing of a certain address reoccurs within a relatively short time interval
  - Spatial locality
    - Property whereby data accessed within a certain time interval is distributed among relatively nearby addresses

Memory Hierarchy (2/2)

- These tendencies often apply to business computations and other non-numeric computations, but are not generally applicable to numeric computation programs.
- Especially in large-scale scientific computations, there is often no temporal locality for data references.
- This is a major reason why vector-type supercomputers are advantageous for scientific computations.

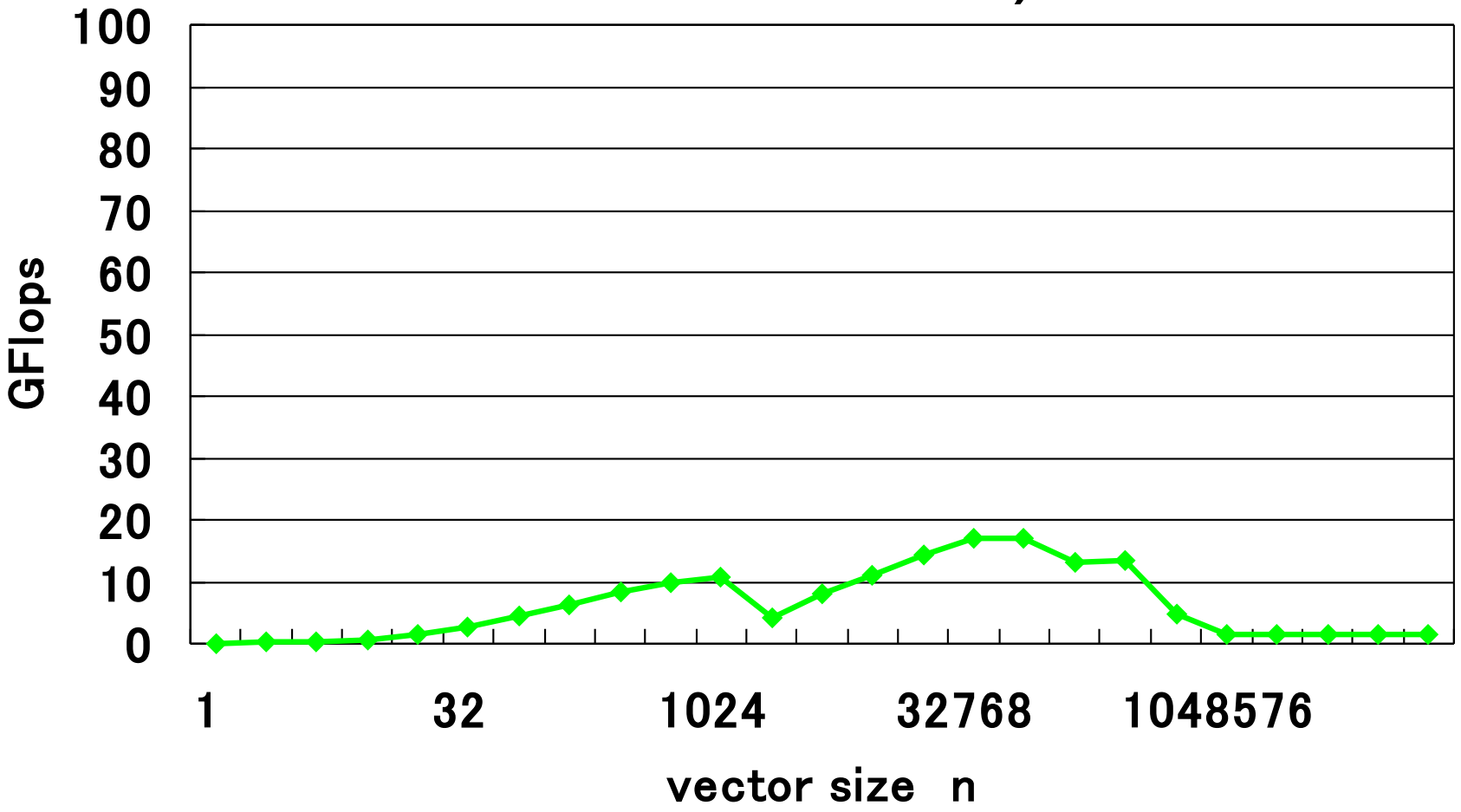
Concept of Byte/Flop

- The amount of memory access needed when performing a single floating-point operation is defined in byte/flop.

```
void daxpy(int n, double a, double *x, double *y)
{
    int i;
    for (i = 0; i < n; i++)
        y[i] += a * x[i];
}
```

- With daxpy, double-precision real-number data must be loaded/stored three times (24 bytes total) in order to perform two double-precision floating-point operations per single iteration.
  - In this case, 24Byte/2Flop = 12Byte/Flop.
- The smaller the Byte/Flop value is better.

Performance of DAXPY (Intel Xeon E3-1230 3.2GHz 8MB L3 cache, Intel MKL 10.3)



PC and Vector-type Supercomputer Memory Bandwidth

- Intel Xeon E5-2687W (Sandy Bridge-EP 3.1GHz, 4 x DDR3-1600, 2 sockets/node)
  - The theoretical peak performance of each node is 24.8GFlops×8 cores×2 sockets=396.8GFlops
  - Memory bandwidth up to 51.2GB/s
  - Byte/Flop value is 51.2/396.8≒0.129
- NEC SX-9A (16 CPUs/node)
  - The theoretical peak performance of each node is 102.4GFlops×16CPU=1638.4GFlops
  - Memory bandwidth up to 4TB/s
  - Byte/Flop value is 4096/1638.4=2.5

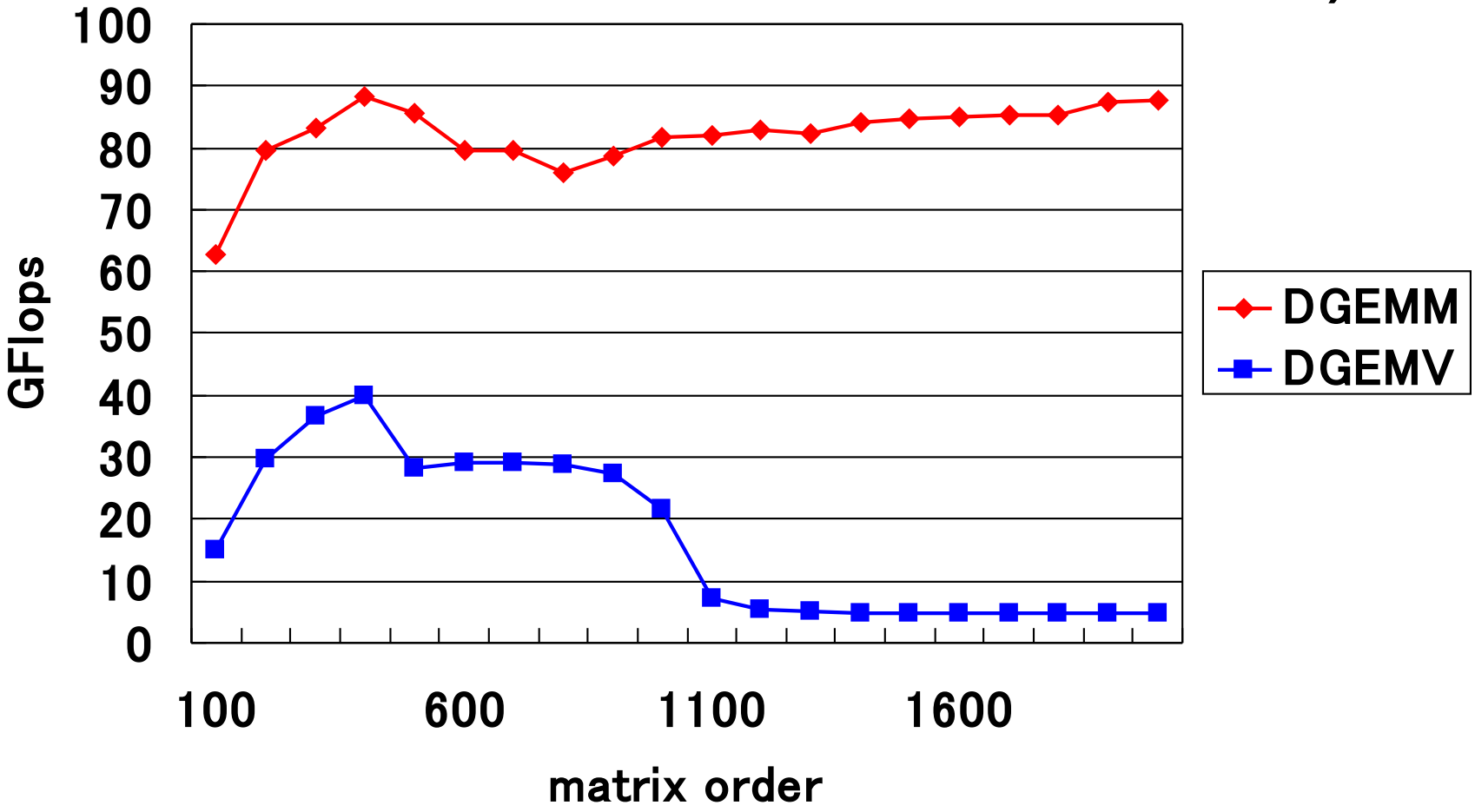
Comparison of Theoretical Performance in DAXPY

- Intel Xeon E5-2687W
  - Theoretical peak performance of each node: 396.8GFlops
  - In the case where the working set exceeds the cache capacity, the memory bandwidth (51.2GB/s) is rate-limiting and so the limit is (51.2GB/s)/(12Byte/Flop)≒4.27GFlops
  - Only approximately 1.1% of theoretical peak performance!
- NEC SX-9A
  - Theoretical peak performance of each node: 1,638.4GFlops
  - The memory bandwidth (4TB/s) is rate-limiting, and so the limit is (4TB/s)/(12Byte/Flop)≒341.3GFlops
  - Approximately 20.8% of theoretical peak performance.

Arithmetic Operations in BLAS

BLAS	Loads + Stores	Operations	Ratio $n = m = k$
Level 1 DAXPY $y = y + \alpha x$	$3n$	$2n$	$3:2$
Level 2 DGEMV $y = \beta y + \alpha Ax$	$mn + n + 2m$	$2mn$	$1:2$
Level 3 DGEMM $C = \beta C + \alpha AB$	$2mn + mk + kn$	$2mnk$	$2:n$

Performances of DGEMV and DGEMM (Intel Xeon E3-1230 3.2GHz 8MB L3 cache, Intel MKL 10.3)



Significance of Performance Tuning

- In the case of calculations whose runtime lasts for several months or longer, optimization may result in a reduction of runtime on the order of a month.
- As in the case of numeric libraries, if a program is used by many people, tuning will have sufficient value.
- If tuning results in a 30% improvement in performance, for example, the net result is the same as using a machine having 30% higher performance.

Optimization Policy

- If available, use a vendor-supplied high-speed library as much as possible.
  - BLAS, LAPACK, etc.
- The optimization capability of recent compilers is extremely high.
- Optimization that can be performed by the compiler must not be performed on the user side.
  - Requires extra effort
  - Results in a program that is complicated and may contain bugs
- Overestimates the optimizing capability of compilers
  - Humans are dedicated to improving algorithms.
  - Unless otherwise unavoidable, do not use an assembler.

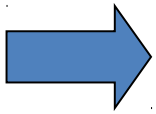
Optimization Information of Fujitsu Fortran Compiler

```
(line-no.)(nest)(optimize)
      80                !$OMP DO
      81    1   p             DO 70 II=1,NX,NBLK
      82    2   p               DO 30 JJ=1,NY,NBLK
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    PREFETCH      : 2
                        <<<      A: 2
                        <<< Loop-information  End >>>
      83    3   p                 DO 20 I=II,MIN0(II+NBLK-1,NX)
      84    3                  !OCL SIMD(ALIGNED)
                        <<< Loop-information Start >>>
                        <<<  [OPTIMIZATION]
                        <<<    SIMD
                        <<<    SOFTWARE PIPELINING
                        <<< Loop-information  End >>>
      85    4   p   8v              DO 10 J=JJ,MIN0(JJ+NBLK-1,NY)
      86    4   p   8v                B(J,I-II+1)=A(I,J)
      87    4   p   8v        10      CONTINUE
```

Loop Unrolling (1/2)

- Loop unrolling expands a loop in order to do the following:
  - Reduce loop overhead
  - Perform register blocking
- If expanded too much, register shortages or instruction cache misses may occur, and so care is needed.

```
double A[N], B[N], C;
for (i = 0; i < N; i++) {
  A[i] += B[i] * C;
}
```

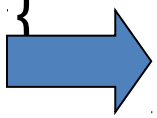


```
double A[N], B[N], C;
for (i = 0; i < N; i += 4) {
  A[i] += B[i] * C;
  A[i+1] += B[i+1] * C;
  A[i+2] += B[i+2] * C;
  A[i+3] += B[i+3] * C;
}
```

Loop Unrolling (2/2)

```
double A[N][N], B[N][N],
        C[N][N], s;
for (j = 0; j < N; k++) {
  for (i = 0; i < N; j++) {
    s = 0.0;
    for (k = 0; k < N; k++) {
      s += A[i][k] * B[j][k];
    }
    C[j][i] = s;
  }
}
```

Matrix multiplication



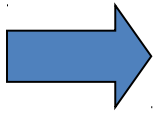
```
double A[N][N], B[N][N],
        C[N][N], s0, s1;
for (j = 0; j < N; j += 2)
  for (i = 0; i < N; i++) {
    s0 = 0.0;  s1 = 0.0;
    for (k = 0; k < N; k++) {
      s0 += A[j][k] * B[j][k];
      s1 += A[j+1][k] * B[j][k];
    }
    C[j][i] = s0;
    C[j+1][i] = s1;
  }
```

Optimized matrix multiplication

Loop Interchange

- Loop interchange is a technique mainly for reducing the adverse effects of large-stride memory accesses.
- In some cases, the compiler judges the necessity and performs loop interchanges.

```
double A[N][N], B[N][N], C;
for (j = 0; j < N; j++) {
  for (k = 0; k < N; k++) {
    A[k][j] += B[k][j] * C;
  }
}
```



```
double A[N][N], B[N][N], C;
for (k = 0; k < N; k++) {
  for (j = 0; j < N; j++) {
    A[k][j] += B[k][j] * C;
  }
}
```

Before loop interchange

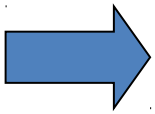
After loop interchange

Padding

- Effective in cases where multiple arrays have been mapped to the same cache location and thrashing occurs
  - Especially in the case of an array having a size that is a power of two
- It is recommended to change the defined sizes of two-dimensional arrays.
- In some instances, this can be handled by specifying the compile options.

```
double A[N][N], B[N][N];
for (k = 0; k< N; k++) {
  for (j = 0; j < N; j++) {
    A[j][k] = B[k][j];
  }
}
```

Before padding



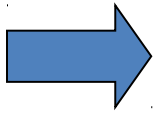
```
double A[N][N+1], B[N][N+1];
for (k = 0; k < N; k++) {
  for (j = 0; j < N; j++) {
      A[j][k] = B[k][j];
  }
}
```

After padding

Cache Blocking (1/2)

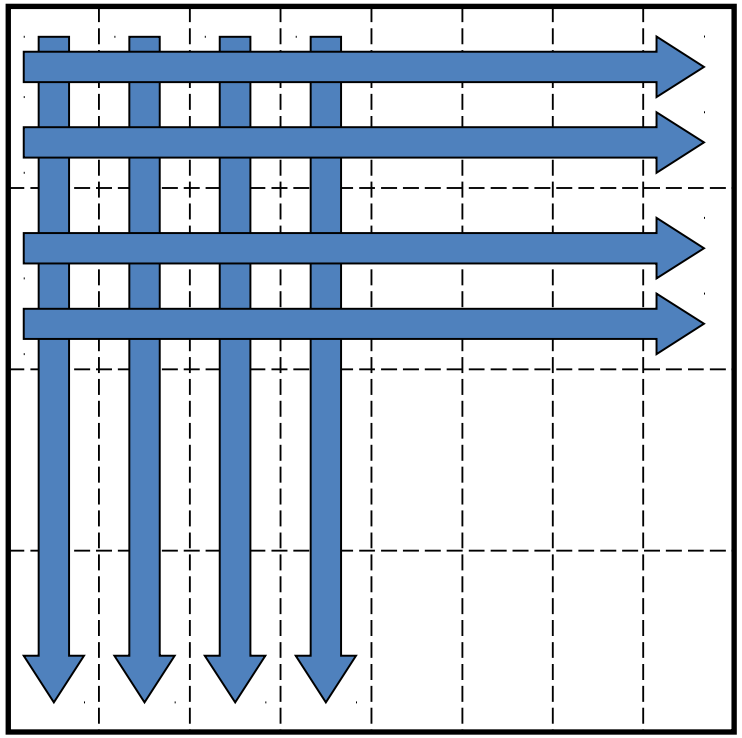
- Effective method for optimizing memory accesses
- Cache misses are reduced as much as possible.

```
double A[N][N], B[N][N], C;
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] += B[j][i] * C;
  }
}
```

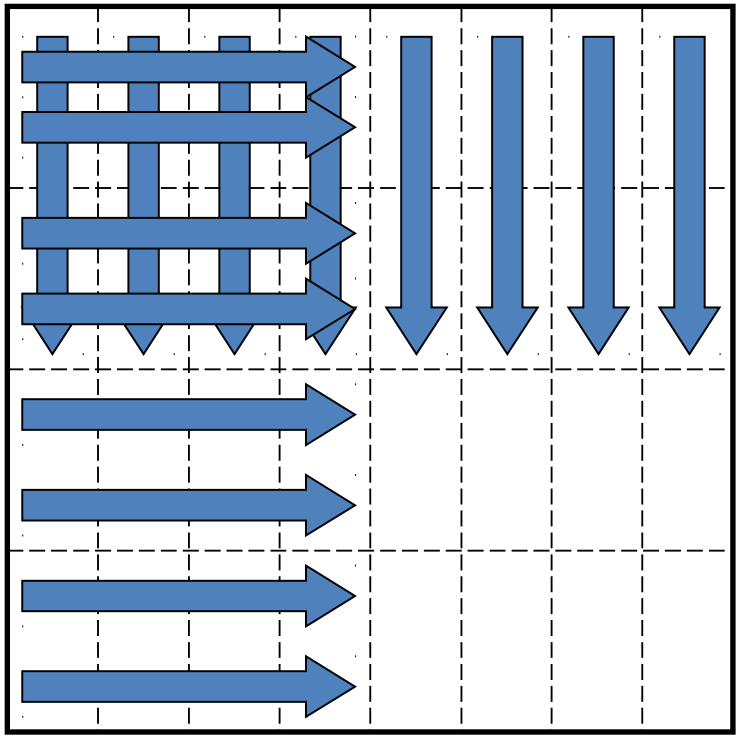


```
double A[N][N], B[N][N], C;
for (ii = 0; ii < N; ii += 4) {
  for (jj = 0; jj < N; jj += 4) {
    for (i = ii; i < ii + 4; i++) {
      for (j = jj; j < jj + 4; j++) {
        A[i][j] += B[j][i] * C;
      }
    }
  }
}
```

Cache Blocking (2/2)



Memory access pattern without blocking



Memory access pattern with blocking

An Implementation of Parallel 1-D FFT Using AVX Instructions on Multi-Core Processors

- The fast Fourier transform (FFT) is an algorithm widely used today in science and engineering.
- Today, a number of processors have short vector SIMD instructions, e.g.,
  - Intel: SSE, SSE2, SSE3, SSSE3, SSE4 and AVX
  - AMD: 3DNow!
  - Motorola: AltiVec
- These instructions provide substantial speedup for digital signal processing applications.
- Efficient FFT implementations with short vector SIMD instructions have also been investigated.

Background

- Many FFT algorithms work well when the data sets fit into the cache.
- However, when the problem size exceeds the cache size, the performance of these FFT algorithms decreases dramatically.
- The key issue in the design of large FFTs is minimizing the number of cache misses.
- Thus, both vectorization and high cache utilization are particularly important with respect to high performance on processors that have short vector SIMD instructions.

Related Works

- FFTW 3.3 [Frigo and Johnson]
  - Supports AVX instructions (new in version 3.3.).
  - The recursive call is employed to access main memory hierarchically.
  - http://www.fftw.org/
- SPIRAL [Pueschel et al.]
  - Supports AVX instructions.
  - The goal of SPIRAL is to push the limits of automation in software and hardware development and optimization for DSP algorithms.
  - http://www.spiral.net/

Approach

- Some previously presented six-step FFT algorithms [VanLoan92] separate the multicolumn FFTs from the transpositions.
- Taking the opposite approach, we combine the multicolumn FFTs and transpositions to reduce the number of cache misses.
- We modify the conventional six-step FFT algorithm to reuse data in the cache memory.
  → We will call it a "block six-step FFT".

Intel AVX Instructions

- Intel Advanced Vector Extensions (AVX) were introduced into the Sandy Bridge processor.
- The most direct way to use the AVX instructions is to insert the assembly language instructions inline into source code.
- However, this can be time-consuming and tedious, and assembly language inline programming is not supported on all compilers.
- The Intel C/C++ and Fortran Compilers support automatic vectorization of floating-point loops using AVX instructions.

How to Use the AVX Instructions

- The AVX instructions may be used in the following ways.

  (1) Vectorization by compiler

  (2) Using AVX intrinsic functions

  (3) Using an inline assembler

  (4) Directly writing a “.s” file with an assembler

- In order from (1) to (4), the coding increases in complexity, but there are advantages from the perspective of performance.

An Example of a Vectorizable Radix-2 FFT Kernel

```
 SUBROUTINE FFT(A,B,W,M,L)
COMPLEX*16 A(M,L,*),B(M,2,*),W(*)
COMPLEX*16 C0,C1
DO J=1,L
!DIR$ VECTOR ALIGNED
    DO I=1,M
      C0=A(I,J,1)
      C1=A(I,J,2)
      B(I,1,J)=C0+C1
      B(I,2,J)=W(J)*(C0-C1)
    END DO
END DO
RETURN
END
```

The innermost loop lengths are varied from 1 to n/2 for n-point FFTs during log2(n) stages.

First Stage of a Vectorizable Radix-2 FFT Kernel

```
      SUBROUTINE FFT1ST(A,B,W,L)
     COMPLEX*16 A(L,*),B(2,*),W(*)
     COMPLEX*16 C0,C1
!DIR$ VECTOR ALIGNED
     DO J=1,L
        C0=A(J,1)
        C1=A(J,2)
        B(1,J)=C0+C1
        B(2,J)=W(J)*(C0-C1)
     END DO
     RETURN
     END
```

When the innermost loop length is one, the double-nested loop can be collapsed into a single-nested loop to expand innermost loop length.

Vectorized Assembly Code of Radix-2 FFT Kernel

```
vmovupd   (%edx,%ecx), %ymm3
vmovupd   (%edx,%ebx), %ymm4
vsubpd    %ymm4, %ymm3, %ymm5
vaddpd    %ymm4, %ymm3, %ymm2
vmulpd    %ymm5, %ymm1, %ymm7
vmovupd   %ymm2, (%edx,%edi)
vshufpd   $5, %ymm5, %ymm5, %ymm6
vmulpd    %ymm6, %ymm0, %ymm2
vmovupd   32(%edx,%ecx), %ymm5
vmovupd   32(%edx,%ebx), %ymm6
```

```
vaddsubpd %ymm2, %ymm7, %ymm3
vsubpd    %ymm6, %ymm5, %ymm7
vaddpd    %ymm6, %ymm5, %ymm4
vmovupd   %ymm3, (%edx,%eax)
vmulpd    %ymm7, %ymm1, %ymm2
vmovupd   %ymm4, 32(%edx,%edi)
vshufpd   $5, %ymm7, %ymm7, %ymm7
vmulpd    %ymm7, %ymm0, %ymm3
vaddsubpd %ymm3, %ymm2, %ymm4
vmovupd   %ymm4, 32(%edx,%eax)
```

Discrete Fourier Transform (DFT)

- 1-D DFT is given by

$$y(k) = \sum_{j=0}^{n-1} x(j)\omega_n^{jk}$$

$$0 \le k \le n-1,\ \ \omega_n = \exp(-2\pi i / n)$$

2-D Formulation

- If n has factors n_1 and n_2 then

$$j = j_1 + j_2 n_1 \quad j_1 = 0, 1, \cdots, n_1 - 1 \quad j_2 = 0, 1, \cdots, n_2 - 1$$

$$k = k_2 + k_1 n_2 \quad k_1 = 0, 1, \cdots, n_1 - 1 \quad k_2 = 0, 1, \cdots, n_2 - 1$$

- Using the above expression, the DFT formulation can be rewritten as follows:

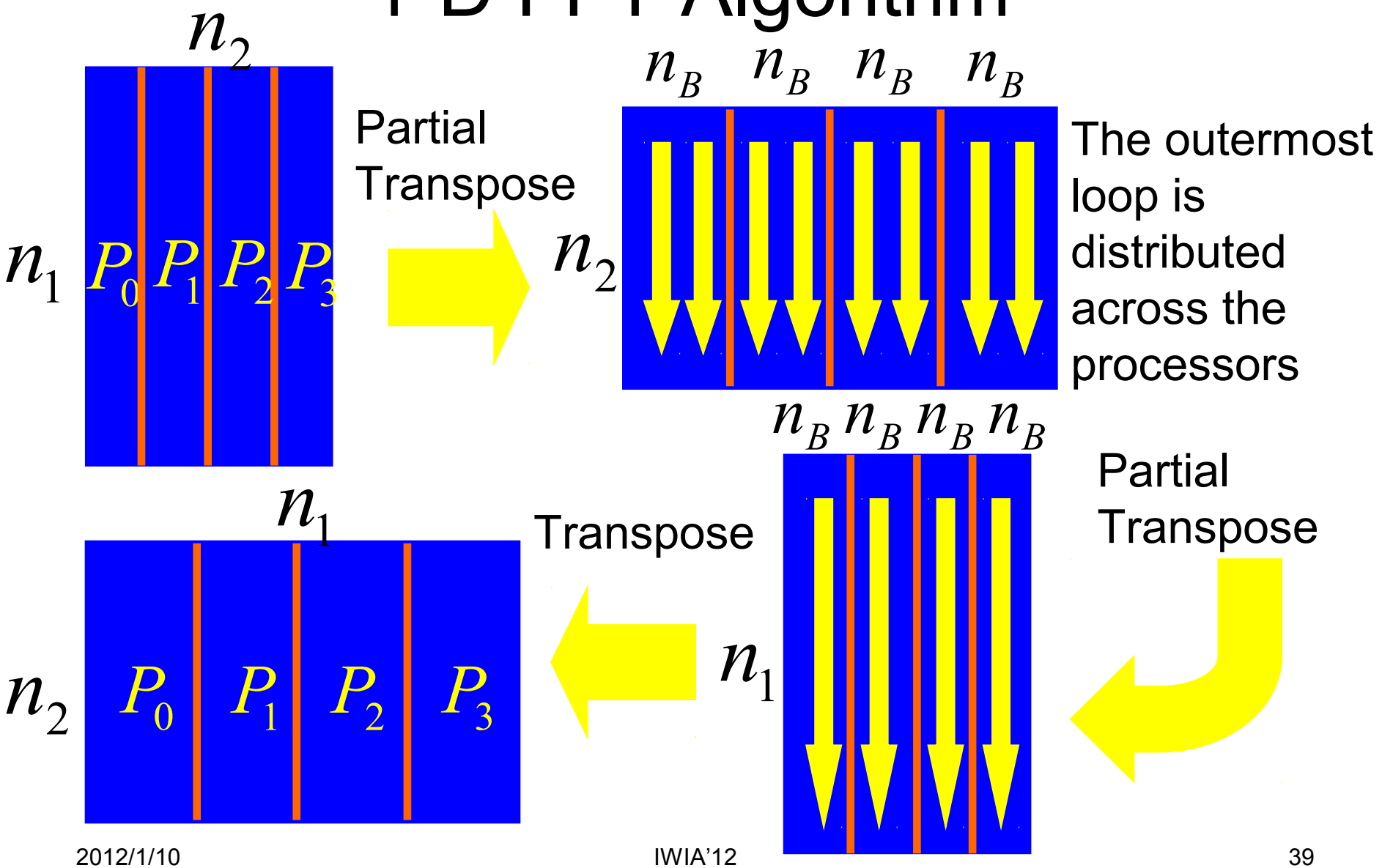
$$y(k_2, k_1) = \sum_{j_1=0}^{n_1-1} \left[\sum_{j_2=0}^{n_2-1} x(j_1, j_2) \omega_{n_2}^{j_2 k_2} \omega_{n_1 n_2}^{j_1 k_2} \right] \omega_{n_1}^{j_1 k_1}$$

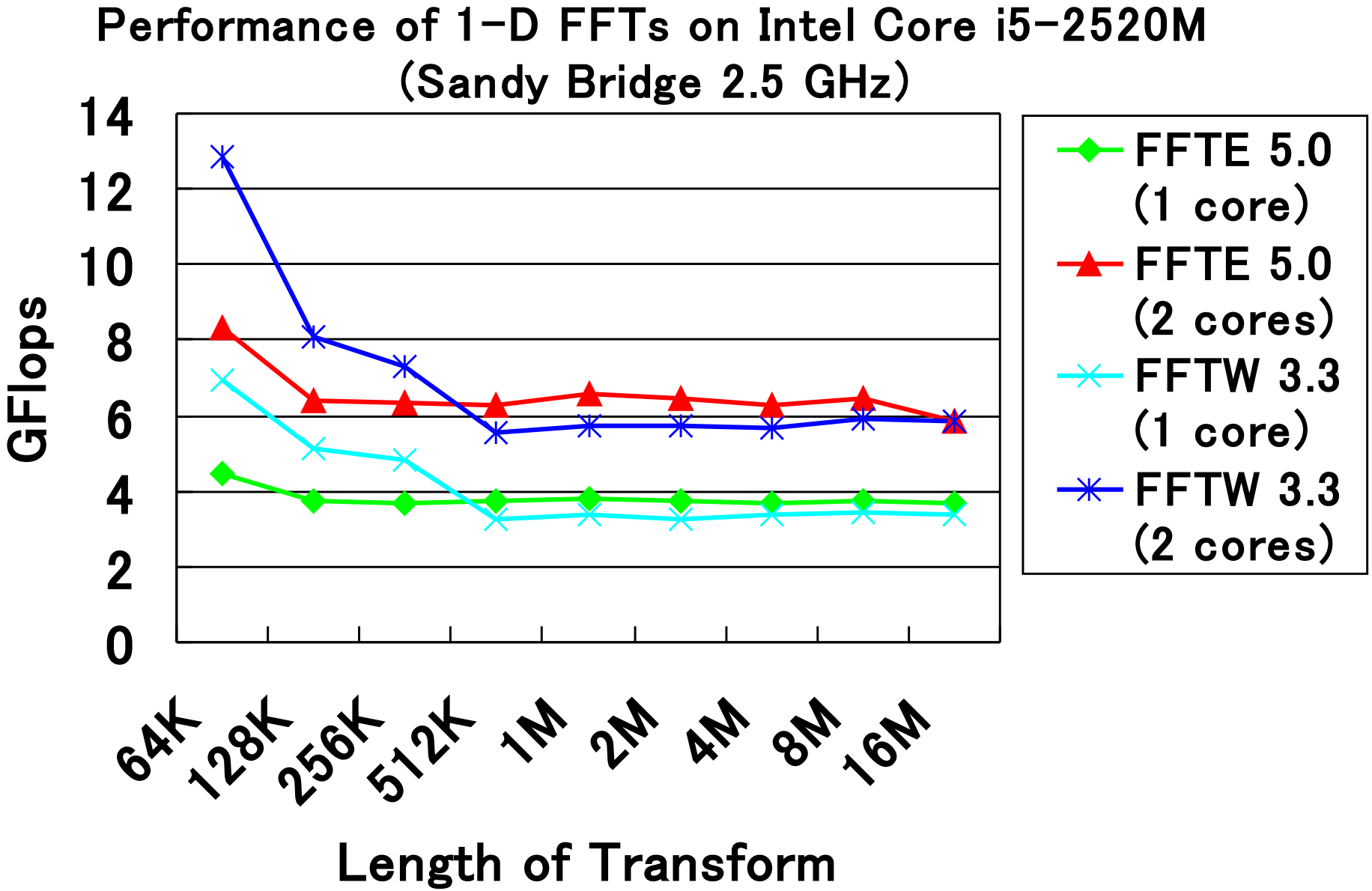
- An n-point FFT can be decomposed into an n_1-point FFT and an n_2-point FFT.

Six-Step FFT Algorithm

- This derivation leads to the following six-step FFT algorithm [Bailey90, VanLoan92]:
- Step 1: Transpose
- Step 2: n_1 individual n_2-point multicolumn FFTs
- Step 3: Twiddle factor ($\omega_{n_1 n_2}^{j_1 k_2}$) multiplication
- Step 4: Transpose
- Step 5: n_2 individual n_1-point multicolumn FFTs
- Step 6: Transpose

Block Six-Step FFT-Based Parallel 1-D FFT Algorithm




Performance of 1-D FFTs on Intel Core i5-2520M
(Sandy Bridge 2.5 GHz)
GFlops
14
12
10
8
6
4
2
0
64K
128K
256K
512K
1M
2M
4M
8M
16M
Length of Transform
FFTE 5.0 (1 core)
FFTE 5.0 (2 cores)
FFTW 3.3 (1 core)
FFTW 3.3 (2 cores)

Conclusion

- Cache-aware algorithms are indispensable for achieving high performance on cache-based processors.
- The ability to perform optimization without the memory bandwidth becoming rate-limited is important for future processors.

Challenges

- To abstract the hardware configuration for application developers is much more important.
- Domain specific-language is one of the solutions.
  - SPIRAL, etc.
- The another way is developing numerical libraries to exploit the system performance.
- How to reduce the cost of performance tuning?
  - Automatic tuning
  - Automated code generation