CCS HPC Winter Seminar
High Performance Parallel Computing Technology for Computational Sciences

"Computation Optimization"

Daisuke Takahashi
Center for Computational Sciences
University of Tsukuba

Contents of Lecture

- What is performance tuning?
- Program optimization methods
  - Register blocking
  - Cache blocking
  - Use of streaming SIMD instructions
- Performance evaluation
  - Examples of benchmark programs

Performance Tuning

- Everyone recognizes the importance of performance in application programs.
- Performance tuning, however, tends to get put off during the software development cycle, and it is never considered in some cases.
- Factors that lead to this type of situation include the following:
  - Recognition that applications can be optimized with only code generation tools and a compiler
  - Unrealistic expectation that the mere use of the latest processor will result in optimal performance while the application is running

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 numerical 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

- Optimization targets many things.
  - Reduction of the amount of code
  - Reduction of the amount of data
  - Reduction of the amount of runtime
- Here, the act of overwriting a program to reduce the runtime is called “optimization”.

Benefits of Optimization

- Optimization reduces the runtime and has the following benefits:
  - More effective use of the computer
  - Lower energy costs
  - More calculations can be performed within the same time
- In consideration of the time required to write and run a program, the longer the runtime of a program, the greater the benefit from optimization.
  - If optimization results in a 30% improvement in performance, for example, the net result is the same as using a machine having 30% higher performance.
- Optimizing a program that will only be run once and that has a short runtime would be rather meaningless.

Prior to Optimizing

- Is there a need to optimize?
- Is the algorithm in use optimal?
- There is no point in optimizing an inefficient algorithm.
  - A bubble sort program, even if optimized, will not be as fast as a quick sort program.
- The optimal algorithm depends largely on the following:
  - Properties of the problem to be solved
  - Architecture, amount of memory, etc., of the computer to be used

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.
- The 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.

First Step in Optimizing

- First, determine the computing performance of one's own program.
- FLOPS (Floating Operations Per Second) is used as a measure of computing performance.
  - Units indicating the number for floating-point operations that can be performed per second
  - MFLOPS (10^6), GFLOPS (10^9), TFLOPS (10^12), PFLOPS (10^15)
- The FLOPS value is computed from the total (or partial) program runtime and the number of operations and is compared to the theoretical peak performance of the processor.
  - In the case of the latest Intel Core i7, the FLOPS value is 32 times the clock.

Time Measurement

- Targets for time measurement are as follows:
  - Elapsed time
  - CPU time
- If the program to be measured has a short runtime, the timer accuracy may be insufficient.
  - Execute an external loop several times and measure.
- In this case, note that the loop may not operate properly as a result of the compiler optimization.
  - Insert a dummy routine or make the measurement target a subroutine and compile separately.

Hot Spots

- The part of a program that accounts for the majority of the computation time is called a “hot spot”.
- First, find out where hot spots exist.
- A profiler is a convenient tool.
  - With Linux, the gprof command can be used.
- As with “gcc –pg foo.c”, by attaching the “-pg” compiler option, special code that writes the profile information used by gprof will be generated.
  - By running a.out, and then specifying gprof a.out, hot spots can be identified.

gprof Output Example

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
48.90	2.90	2.90	2	1.45	2.83	zfft1d0_
32.38	4.82	1.92	49152	0.00	0.00	fft8b_
14.17	5.66	0.84	16384	0.00	0.00	fft8a_
4.55	5.93	0.27	1	0.27	5.93	MAIN__
0.00	5.93	0.00	16384	0.00	0.00	fft235_
0.00	5.93	0.00	4	0.00	0.00	factor_
0.00	5.93	0.00	3	0.00	1.89	zfft1d_
0.00	5.93	0.00	2	0.00	0.00	settbl_
0.00	5.93	0.00	1	0.00	0.00	settbls_

gprof Output Example

- As can be seen from the gprof results:
  - There are three hot spots:
    - zfft1d0_
    - fft8b_
    - fft8a_
- These 3 hot spots consume more than 95% of the total runtime.
  - Optimization should be performed focusing on these hot spots.
  - When writing the program, pay attention so that the hot spots are concentrated.
  - If there are many hotspots, much effort will be required to improve the code.
    - Sometimes it is better to rewrite the code from scratch.

Compile Options

- The performance will vary significantly according to the way in which compile options are specified.
- Use the compiler manual as a reference and try various compile options.
  - "-fast", "-O3", "-O2", etc.
  - With an Intel Compiler, "-xCORE-AVX512" (for latest Core i7)
- Setting a high level of optimization does not necessarily produce faster code.
  - The compiler may optimize excessively.
  - Note that the calculated results may be inconsistent in some cases.

Compiler Directives

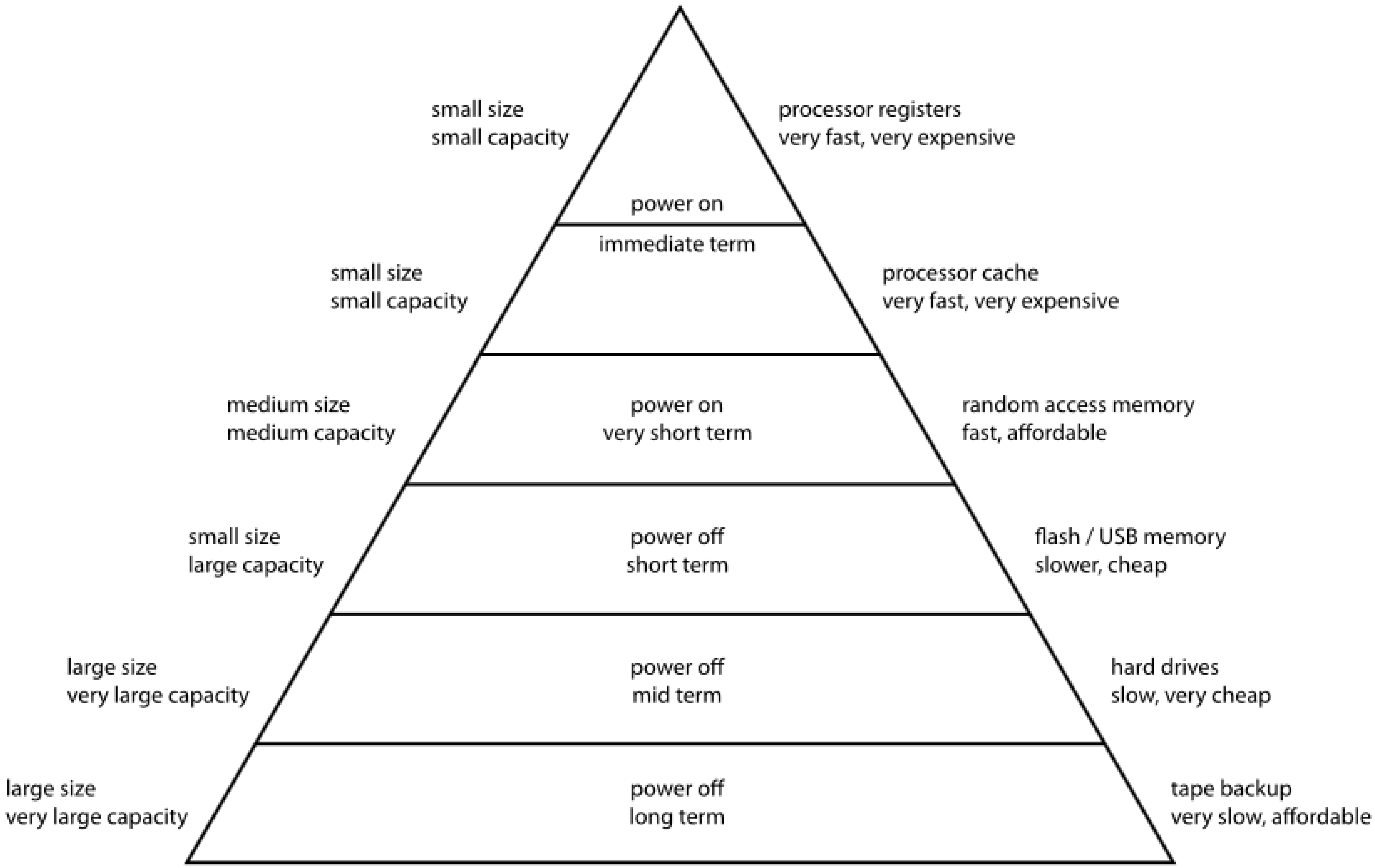
- Compiler directives communicate the intent of the programmer to the compiler and support optimization.
  - Different from compile options, compiler directives allow optimization to be controlled for individual loops.
- Examples of directives
  - When performing vectorization, inform the compiler that there is no loop dependency.
  - Suppress vectorization.
- Often coded in C language as “#pragma”, in Fortran as “!dir$” or “cpgi$l”, etc.

  (Note that the coding may differ according to the compiler.)

Considerations When Writing Programs

- Preserve C or Fortran syntax precisely.
  - With some compilers, only warnings may be output, but these often lead to bugs.
- Compiler-dependent extensions, with the exception of unavoidable circumstances (in the case of a directive, for example), should not be used.
  - Automatic array assignment in GFortran
    - Case such as real*8 a(n), where a(n) is not a dummy argument and n is a variable
  - Program portability deteriorates.
  - Cause of unexpected errors
- To the extent possible, avoid using functions and features that are (thought to be) seldom used.
  - Compiler bugs may not have been removed.

Computer Memory Hierarchy



Source: Wikipedia

Memory Hierarchy (1/3)

- Cost/performance balance in the memory devices that store data
  - "Small capacity × high speed ≈ Large capacity × low speed"
- "Small capacity × high speed" memory devices
  - Registers
- "Large capacity × low speed" memory devices
  - Hard disk and magnetic tape
- "Large capacity × high speed" results in a poor cost/performance balance, and is difficult to implement.

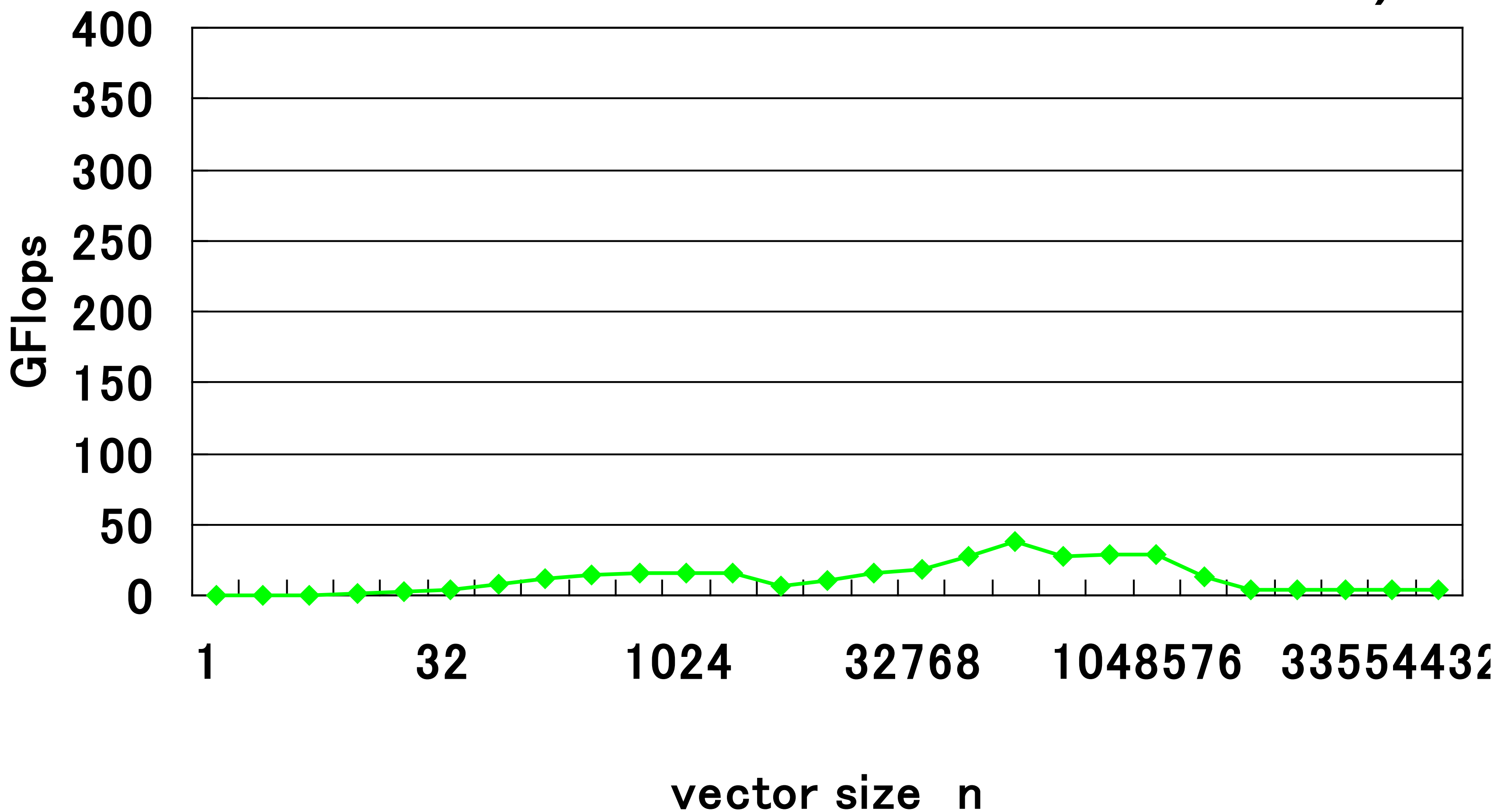
Memory Hierarchy (2/3)

- The 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

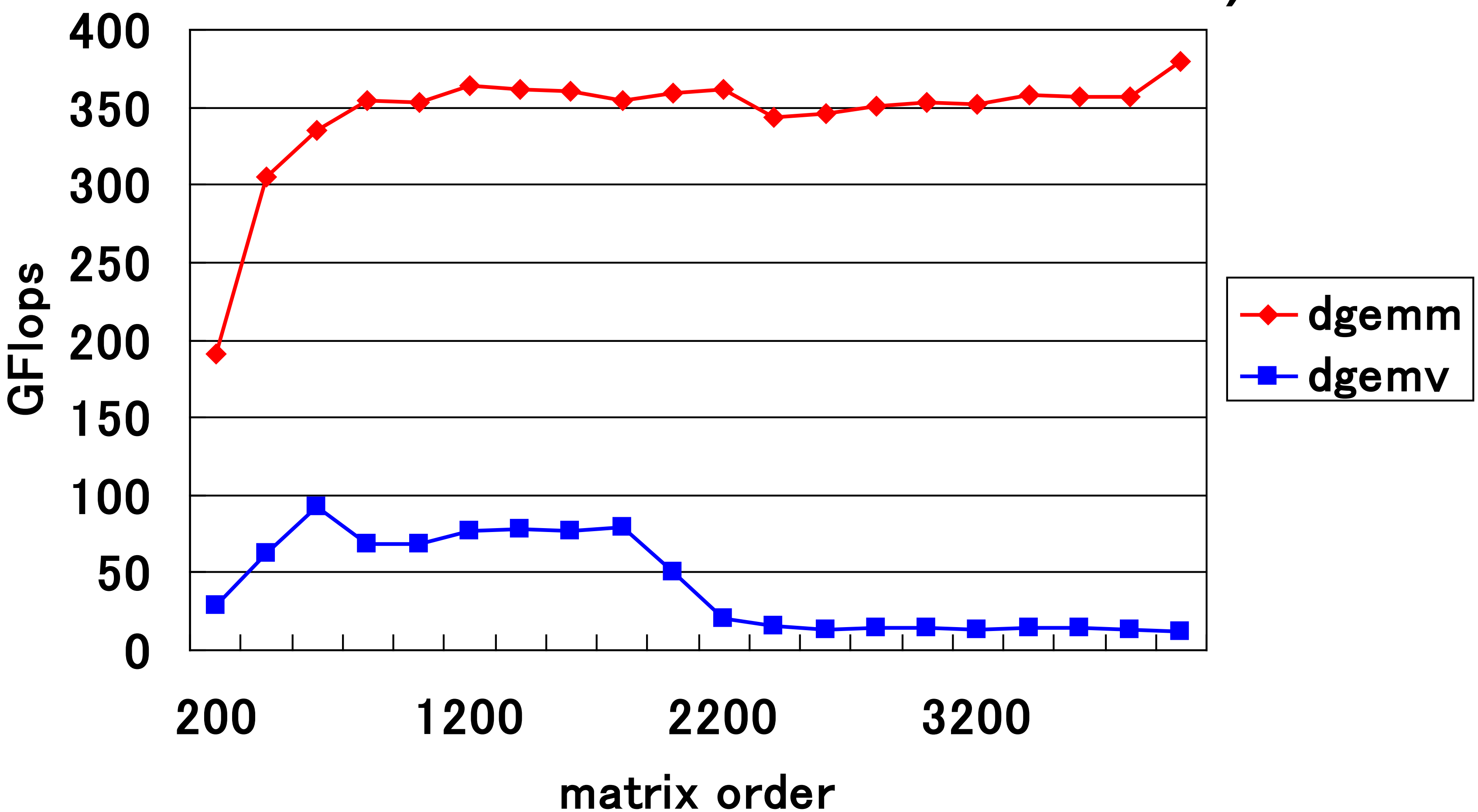
Memory Hierarchy (3/3)

- These tendencies often apply to business computations and other non-numerical computations but are not generally applicable to numerical 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.

Performance of daxpy (Intel Xeon E5-2670 v3 2.3GHz 30MB L3 cache, Intel MKL 11.3)



Performances of dgemv and dgemm (Intel Xeon E5-2670 v3 2.3GHz 30MB L3 cache, Intel MKL 11.3)



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 $C = \beta C + \alpha AB$	$2mn + mk + kn$	$2mnk$	$2:n$

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, the better.

Theoretical Performance in daxpy

- Intel Core i7-7820X (Skylake 3.6GHz, 8 cores, DDR4-2666 x 4)
  - Theoretical peak performance is 115.2GFlops $\times$ 8 cores = 921.6GFlops
  - Maximum memory bandwidth is 85.3GB/s
  - Byte/Flop value is 85.3/921.6 $\approx$ 0.093
- In the case where the working set exceeds the cache capacity, the memory bandwidth (85.3GB/s) is rate-limiting and so the limit is (85.3GB/s)/(12Byte/Flop) $\approx$ 7.1GFlops
- Only approximately 0.8% of theoretical peak performance!

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; j++) {
  for (i = 0; i < N; i++) {
    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[i][k] * B[j][k];
      s1 += A[i][k] * B[j+1][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;
  }
}
```

Before loop interchange

```
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;
  }
}
```

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

Blocking (1/2)

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

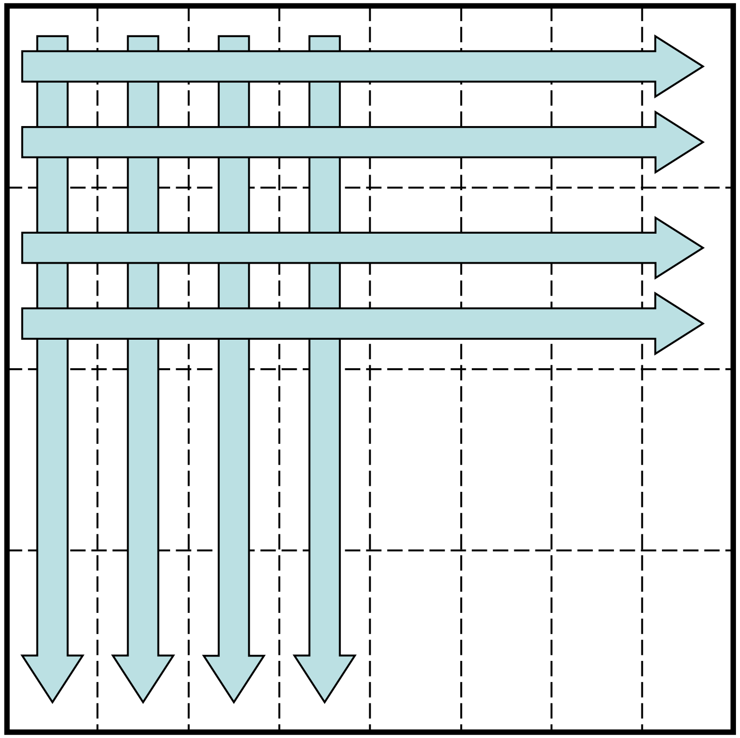
```
double A[N][N], B[N][N], C;
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    A[i][j] += B[j][i] * C;
  }
}
```

Before blocking

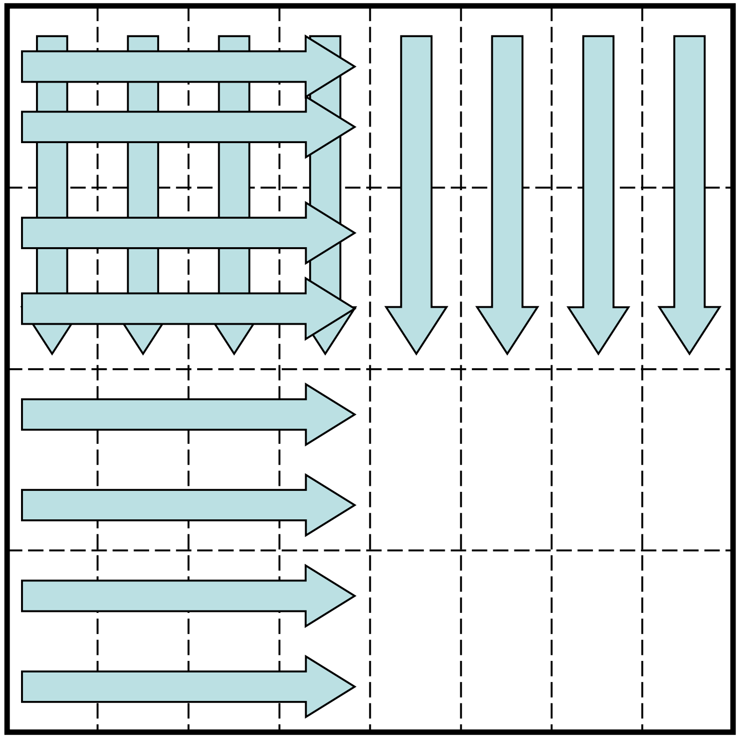
```
double A[N][N], B[N][N], C;
for (i = 0; i < N; i += 4) {
  for (j = 0; j < N; j += 4) {
    for (ii = i; ii < i + 4; ii++) {
      for (jj = j; jj < j + 4; jj++) {
        A[ii][jj] += B[jj][ii] * C;
      }
    }
  }
}
```

After blocking

Blocking (2/2)



Memory access pattern without blocking



Memory access pattern with blocking

Use of Streaming SIMD Instructions

- To process floating-point operations at faster speeds, recent processors are often equipped with what is called streaming SIMD instructions.
  - Intel's SSE/SSE2/SSE3/SSE4/AVX/AVX2/AVX-512 instruction sets
  - AMD Athlon's 3DNow! instruction set
  - Motorola PowerPC's AltiVec instruction set
- With Intel's recent Skylake, the use of AVX-512 instructions enables the floating-point operation performance to be made 32 times as large.

How to Use the SIMD Instruction Set

- The SIMD instruction set may be used in the following ways.

(1) Vectorization by compiler

(2) Using SIMD 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.

Example of calculating product-sum of double-precision complex numbers (a + b * c) with an SSE3 intrinsic function

```
#include <pmmintrin.h>      /* Header file for SSE3 instruction */

static __inline  __m128d ZMULADD(__m128d a, __m128d b, __m128d c)
{
  __m128d br, bi;                          /* 128bit data type */

  br = _mm_movedup_pd(b);                  /* br = [b.r b.r]  real part */
  br = _mm_mul_pd(br, c);                  /* br = [b.r*c.r b.r*c.i] */
  a = _mm_add_pd(a, br);                   /* a = [a.r+b.r*c.r a.i+b.r*c.i] */
  bi = _mm_unpackhi_pd(b, b);              /* bi = [b.i b.i]  imaginary part */
  c = _mm_shuffle_pd(c, c, 1);             /* c = [c.i c.r]  replace real part and
                                                             imaginary part */
  bi = _mm_mul_pd(bi, c);                  /* bi = [-b.i*c.i b.i*c.r] */

  return _mm_addsub_pd(a, bi);             /* [a.r+b.r*c.r-b.i*c.i a.i+b.r*c.i+b.i*c.r] */
}
```

ZAXPY written in C language

```
typedef struct { double r, i; } doublecomplex;

void zaxpy(int n, doublecomplex a, doublecomplex *x, doublecomplex *y)
{
  int i;

  if (a.r == 0.0 && a.i == 0.0) return;

#pragma unroll(8)
#pragma vector aligned
  for (i = 0; i < n; i++) {
    y[i].r += a.r * x[i].r – a.i * x[i].i,
    y[i].i += a.r * x[i].i + a.i * x[i].r;
}
```

ZAXPY written in SSE3 Intrinsic Function

```
#include <pmmintrin.h>

typedef struct { double r, i; } doublecomplex;
__m128d ZMULADD(__m128d a, __m128d b, __m128d c);

void zaxpy(int n, doublecomplex a, doublecomplex *x, doublecomplex *y)
{
  int i;
  __m128d a0;

  if (a.r == 0.0 && a.i == 0.0) return;
  a0 = _mm_loadu_pd(&a);
#pragma unroll(8)
  for (i = 0; i < n; i++)
    _mm_store_pd(&y[i], ZMULADD(_mm_load_pd(&y[i]), a0, _mm_load_pd(&x[i])));
}
```

Objective of Performance Evaluation (1/3)

- Upon actually using a computer system, have you ever had the following type of experience?
  - "I thought this would be a high-performance system, but when I tried using it, the actual performance was not as high as I had expected."
- There are two main reasons for this.
  - Although touted as "high performance," the computer system was well suited for a certain type of calculations that differed from the calculations that the user attempted to execute.
  - Actually, the computer system concealed its high performance, and the problem lies with the user's method of usage, which did not elicit high performance.

Objective of Performance Evaluation (2/3)

- There is only one type of computer throughout the world, and unless technical advances are realized in the future, there will not be much need for “performance evaluations”.
  - However, the reality is that there is a proliferation of many different types of processors and computer systems throughout the world.
- The user must determine which computer system will be able to calculate efficiently the types of problems that he or she desires to solve.
- Also, when improving hardware and software to enhance computer performance, in order to “know thyself”, the developers of the computer system must perform a “performance evaluation” and use the results to improve the performance.

Objective of Performance Evaluation (3/3)

- By performing a performance evaluation:
  - A computer system’s level of performance and the type of problems for which it is best suited for solving can be ascertained.
  - Also, the time required for calculations of extra-large problems that are extremely time-consuming can be ascertained in advance.
- In addition, the decision to perform a calculation with a high cost-performance can be made by the user in consideration of both the cost of using the computer system and its performance.

Indicator of Performance Evaluation

- MIPS (Million Instructions Per Second)
  - Expresses the number of millions of instructions that can be executed per second by the CPU
  - MIPS is ultimately a measure of the number of instructions executed and is not suitable for comparisons of performance among computers having different architectures.
- FLOPS (Floating Operations Per Second)
  - Expresses the number of floating-point operations that can be executed per second
  - MFLOPS, GFLOPS, TFLOPS
- SPEC (The Standard Performance Evaluation Corporation)
  - SPEC benchmark values include SPECint, which indicates the integer processing performance, and SPECfp, which indicates the floating-point processing performance.

Examples of Benchmark Programs

- SPEC
- LINPACK
- NAS Parallel Benchmarks (NPB)
- HPC Challenge (HPCC) Benchmark

Overview of Each Benchmark (1/4)

- SPEC (Standard Performance Evaluation Corporation)
  - A non-profit organization funded by major vendors
  - Measurement results published at http://www.spec.org
- SPEC CPU2006: Comprehensive performance evaluation of CPU, memory, and compiler
  - CINT2006 (SPECint): Evaluates integer processing performance
  - CFP2006 (SPECfp): Evaluations floating-point processing performance
- Additionally includes SPEC MPI2007, SPEC OMP2001, etc.

Overview of Each Benchmark (2/4)

- LINPACK
  - 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

Overview of Each Benchmark (3/4)

- NAS Parallel Benchmarks
  - The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers
  - The original eight benchmarks specified in NPB 1 mimic the computation and data movement in CFD applications.

NAS Parallel Benchmarks

- Five kernels
  - IS: Integer Sort, random memory access
  - EP: Embarrassingly Parallel
  - CG: Conjugate Gradient, irregular memory access and communication
  - MG: Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
  - FT: discrete 3D fast Fourier Transform, all-to-all communication
- Three pseudo applications
  - BT: Block Tri-diagonal solver
  - SP: Scalar Penta-diagonal solver
  - LU: Lower-Upper Gauss-Seidel solver

Overview of Each Benchmark (4/4)

- HPC Challenge (HPCC) Benchmark Suite
  - HPC Challenge (HPCC) is a suite of tests that examine the performance of HPC architectures using kernels.
  - The suite provides benchmarks that bound the performance of many real applications as a function of memory access characteristics, e.g.,
    - Spatial locality
    - Temporal locality

HPC Challenge (HPCC) Benchmark

- The HPC Challenge benchmark consists at this time of 7 performance tests:
  - HPL (High Performance Linpack)
  - DGEMM (matrix-matrix multiplication)
  - STREAM (sustainable memory bandwidth)
  - PTRANS (A=A+B^T, parallel matrix transpose)
  - RandomAccess (integer updates to random memory locations)
  - FFT (complex 1-D discrete Fourier transform)
  - b_eff (MPI latency/bandwidth test)

Summary

- To reduce execution time, optimization is important.
  – However, a determination must be made as to whether optimization is really necessary.
- The ability to perform optimization without the memory bandwidth becoming rate-limited is important for future processors.
- Performance evaluations are effective for ascertaining the performance of a computer prior to usage.

Problem 8

- Develop the following programs in arbitrary programming languages:
  - Matrix multiplication with cache blocking
  - Matrix multiplication without cache blocking
- Then, measure the performance (GFlops) of a 1000×1000 double-precision matrix multiplication on your PCs.
- Submit the source codes and performance results.