Vectorization and Parallelization of Loops in C/C++ Code

Xuejun Liang, Ali A. Humos, and Tzusheng Pei
Department of Computer Science, Jackson State University
Jackson, MS 39217, USA

Outlines

A. Introduction
B. Intel SIMD Extensions to IA-32
C. Vectorizing Using Intrinsics
D. Auto-Vectorizing Using Compiler Flags
E. Auto-Parallelization and OpenMP
F. Performance Analysis
G. Conclusion

A: Introduction

- SIMD (Single Instruction Multiple Data) Extension
  - Use large registers to hold multiple data
  - Apply single operation to multiple data simultaneously
- Multicore processor
  - Single computing component with two or more independent actual processing units
- Purpose: How to utilize (using VC++ and GNU g++)
  - SIDM extension instructions (Vectorization)
  - Multiple processing cores (Parallelization)

Introduction: Examples

- Vector Multiplication

Loop A →

```
1  for(int i = 0; i < MAX_DIM; ++i)
2    c[i] = a[i] * b[i];
```

- Matrix Multiplication (stored in Row-Major)

Loop B1 →
Loop B2 →
Loop B3 →

```
1  for(int i = 0; i < n; ++i) {
2    for(int j = 0; j < n; ++j)  {
3      c[i*n+j] = 0;
4      for(int k = 0; k < n; k++) {
5        c[i*n+j] += a[i*n+k] * b[k*n+j];
6      }
7    }
8  }
```

Introduction: Examples (Cont.)

- Matrix Multiplication (Stored in Column-Major)

Loop C1 →
Loop C2 →
Loop C3 →

```
1  for(int i = 0; i < n; ++i) {
2    for(int j = 0; j < n; ++j)  {
3      c[i+n*j] = 0;
4      for(int k = 0; k < n; k++) {
5        c[i+n*j] += a[i+n*k] * b[k+n*j];
6      }
7    }
8  }
```

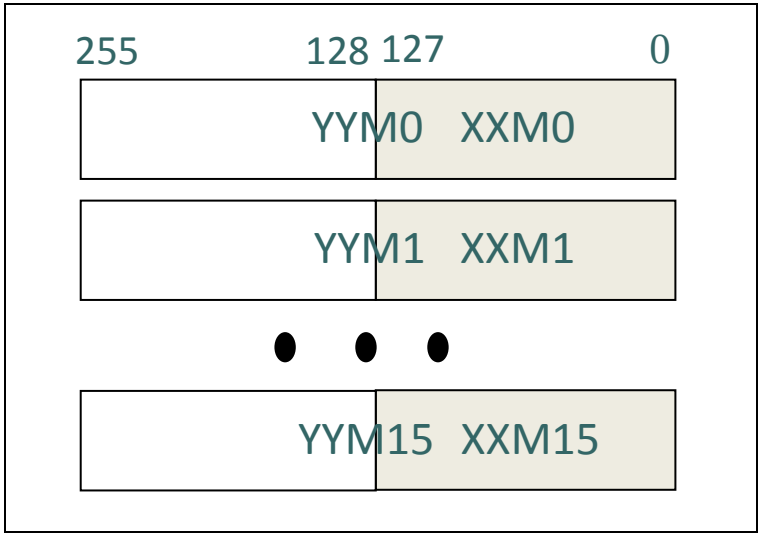
  - Note: David A. Patterson and John L. Hennessy use the column-major matrix multiplication in their famous book to illustrate “going faster”.

B: Intel SIMD Extensions to IA-32

- SIMD (Single Instruction Multiple Data) Extension
  - Use large registers to hold multiple data
  - Apply single operation to multiple data simultaneously
- Intel has experienced SIMD extensions many times
  - Multi-Media Extension (MMX)
  - Streaming SIMD Extension (SSE, SSE2, SSE3, SSE3, SSE4)
  - Advanced Vector Extension (AVX, AVX2, AVX-512)
- AVX is what we work on

Advanced Vector Extension (AVX)

- Register YMM: 256 bits
  - 8 x 32-bit single precision floating-point operations
  - 4 x 64-bit double precision floating-point operations
  - SEE2 instructions use the lower 128 bits



Advanced Vector Extension (AVX)

- Arithmetic AVX Floating Point Instructions
  - ADD[PS/PD/SS/SD] XMM1, XMM2/M128
  - VADD[PS/PD/SS/SD] XMM1, XMM2, XMM3/M128
  - VADD[PS/PD/SS/SD] YMM1, YMM2, YMM3/M256
    - PS: Packed single precision
    - PD: Packed double precision
    - SS: Scalar single precision
    - SD: Scalar double precision
    - M128: memory data aligned at 128-bit boundary
    - M256: memory data aligned at 256-bit boundary

Intel AVX Instrinsics

- The compiler can replace them with the proper assembly instructions
- Three new C data types
  - __m256 is packed float
  - __m256d is packed double
  - __m256i is packed integer

Intel AVX Instrinsics (Cont.)

- AVX intrinsic name format
  - _m256_OP_SURFIX
    - OP is an operation, such as add, sub, etc.
    - SURFIX is one of PS, PD, SS, and SD
- Examples of intrinsic prototypes
  - __m128 _mm_add_ps(__m128 m1, __m128 m2);
  - __m128 _mm256_add_ps(__m128 m2, __m128 m3);
  - __m256 _mm256_add_ps(__m256 m2, __m256 m3);

C. Vectorizing Matrix Multiplication

- Pseodo Code (Row-Major)

```
01  for(int i = 0; i < n; ++i) {
02    for(int j = 0; j < n; j += 8)       {
03      (c[i*n+j+0],...,c[i*n+j+7]) = (0,...,0);
04      for(int k = 0; k < n; k++) {
05        (c[i*n+j+0],...,c[i*n+j+7]) +=
06          (a[i*n+k+0],...,a[i*n+k+0]) *
07          (b[k*n+j+0],...,b[k*n+j+7]);
08      }
09    }
10  }
```

$$(c[i,\ j+0],\ldots,c[i,\ j+7]) = \sum_{k=0}^{n-1} (a[i,k],\ldots,a[i,k]) * (b[k,\ j+0],\ldots,b[k,\ j+7])$$

Vectorizing Using Intrinsics

- Matrix Multiplication (Row-Major)

Loop BV1 →
Loop BV2 →
Loop BV3 →

```
01 for (int i = 0; i < n; i++) {
02   for (int j = 0; j < n; j += 8) {
03     __m256 m0 = _mm256_setzero_ps();
04     for (int k = 0; k < n; k++) {
05       __m256 m1 = _mm256_broadcast_ss(a+i*n+k);
06       __m256 m2 = _mm256_load_ps((b+k*n+j));
07       __m256 m3 = _mm256_mul_ps(m1, m2);
08       m0 = _mm256_add_ps(m0, m3);
09     }
10     _mm256_store_ps(c+i*n+j, m0);
11   }
12 }
```

```
m0 = (c[i*n+j+0], ..., c[i*n+j+7])
m1 = (a[i*n+k+0], ..., a[i*n+k+0])
m2 = (b[k*n+j+0], ..., b[k*n+j+7])
```

Vectorizing Matrix Multiplication

- Pseodo Code (Column-Major)

```
01  for(int i = 0; i < n; i += 8) {
02    for(int j = 0; j < n; ++j) {
03      (c[i+n*j+0],..., c[i+n*j+7]) = (0,...0);
04      for(int k = 0; k < n; k++) {
05        (c[i+n*j+0],..., c[i+n*j+7]) +=
06          (a[i+n*k+0],...,a[i+n*k+7]) *
07          (b[k+n*j+0],...,b[k+n*j+0];
08      }
09    }
10  }
```

$$(c[i+0,\ j],...,c[i+7,\ j]) = \sum_{k=0}^{n-1} (a[i+0,k],...,a[i+7,k]) * (b[k,\ j],...,b[k,\ j])$$

Vectorizing Using Intrinsics

- Matrix Multiplication (Column-Major)

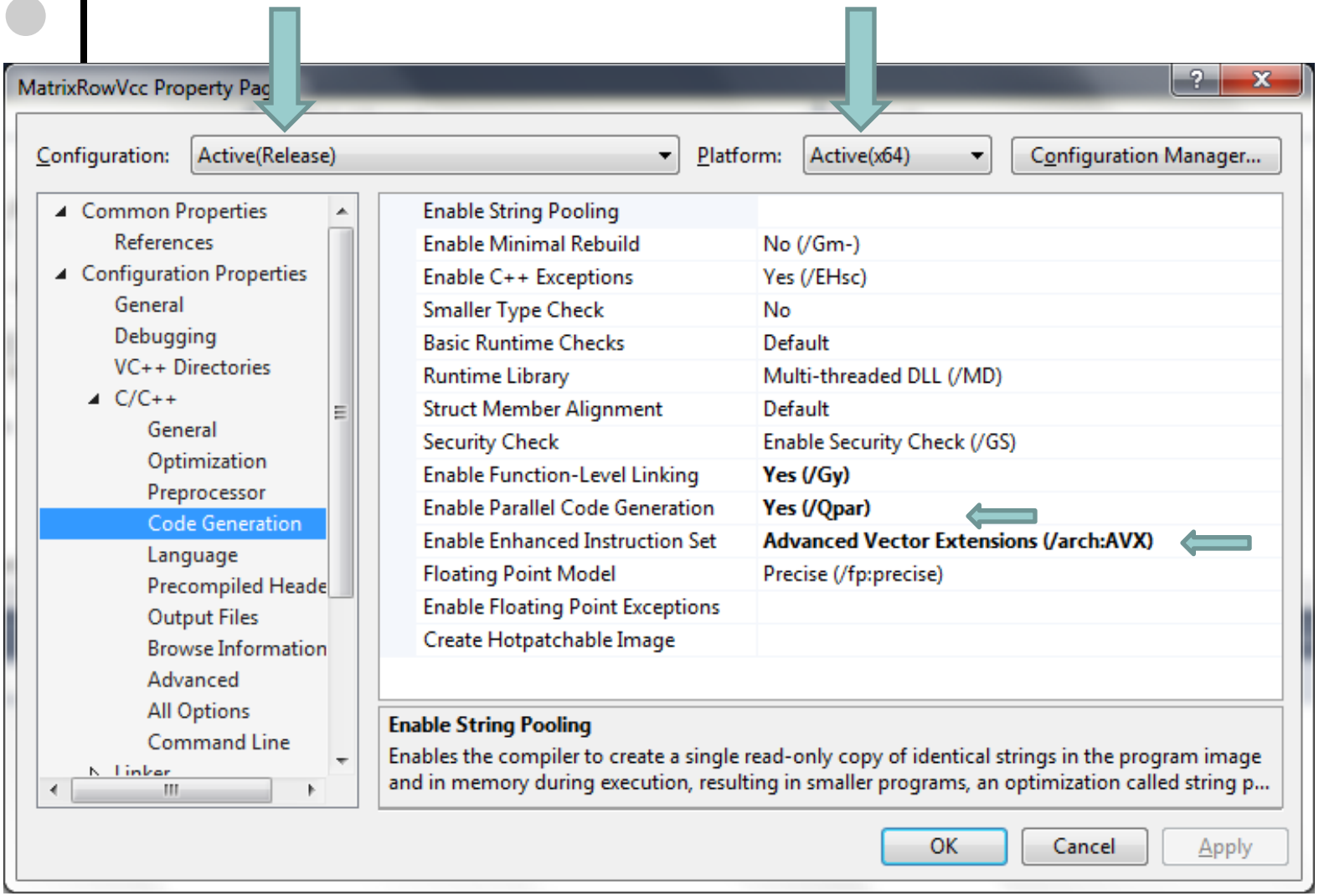
Loop CV1 →
Loop CV2 →
Loop CV3 →

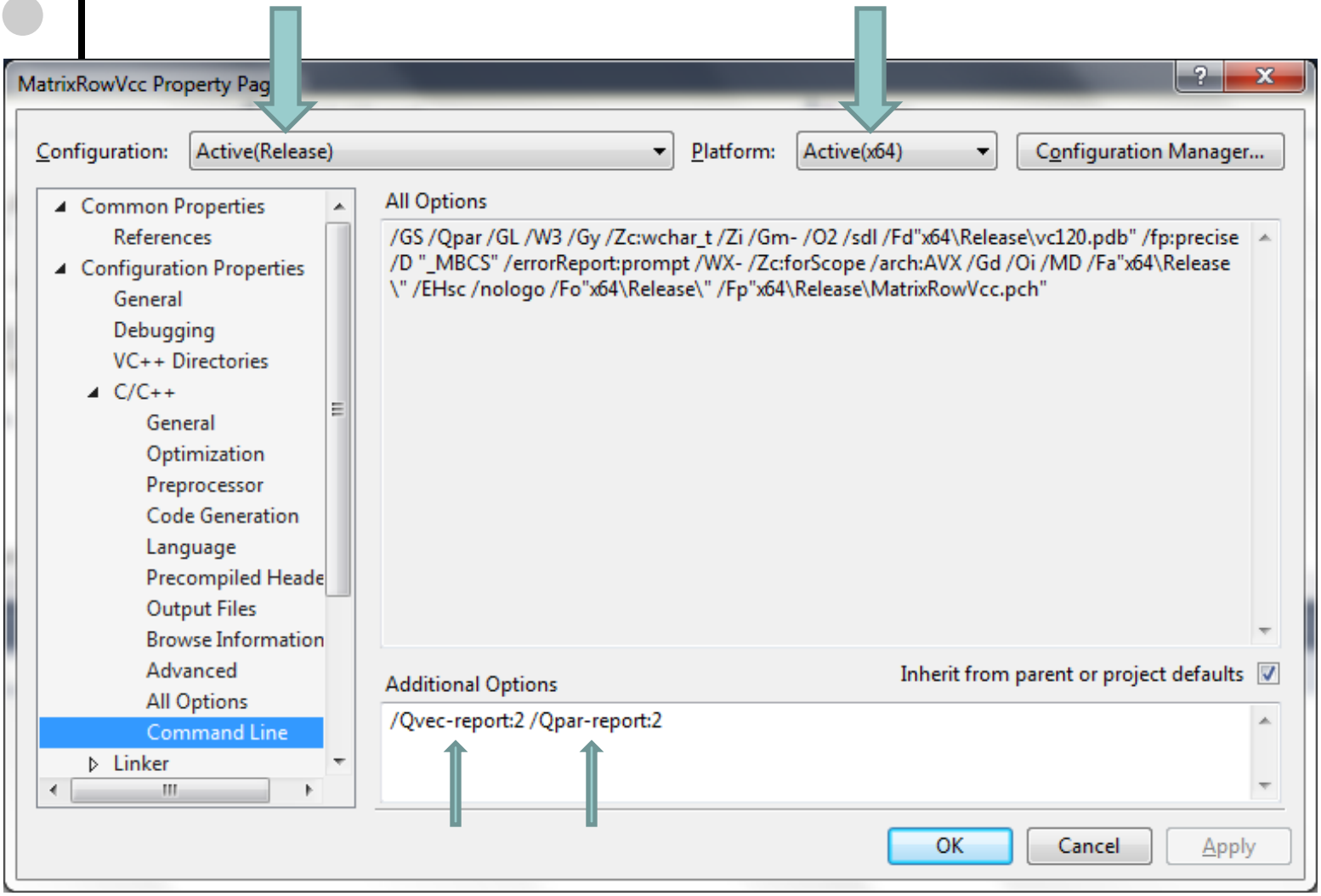
```
01 for ( int i = 0; i < n; i += 8 ) {
02   for ( int j = 0; j < n; j++ ) {
03     __m256 m0 = _mm256_setzero_ps();
04     for( int k = 0; k < n; k++ ) {
05       __m256 m1 = _mm256_load_ps(a+i+n*k);
06       __m256 m2 = _mm256_broadcast_ss(b+k+n*j);
07       __m256 m3 = _mm256_mul_ps(m1, m2);
08       m0 = _mm256_add_ps(m0, m3);
09    }
10     _mm256_store_ps(c+i+n*j, m0);
11   }
12 }
```

```
m0 = (c[i+n*j+0], ..., c[i+n*j+7])
m1 = (a[i+n*k+0], ..., a[i+n*k+7])
m2 = (b[k+n*j+0], ..., b[k+n*j+0])
```

D. Auto-Vectorization (VC++)

- Visual C++ supports auto-vectorization by default
- Select **Release** as the solution configuration and **x64** as the solution platform in your project
- Add the following code generation compiler option to specify a targeted SIMD extension of IA32
  - **/arch:AVX**
- Add the following additional compiler option to get the reason code for results
  - **/Qvec-report:2**
- Add the following line before a loop to disable vectorization
  - **#pragma loop(no_vector)**


MatrixRowVcc Property Pag
Configuration: Active(Release)
Platform: Active(x64)
Configuration Manager...
Common Properties
References
Configuration Properties
General
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Heade
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
Enable String Pooling
Enable Minimal Rebuild
No (/Gm-)
Enable C++ Exceptions
Yes (/EHsc)
Smaller Type Check
No
Basic Runtime Checks
Default
Runtime Library
Multi-threaded DLL (/MD)
Struct Member Alignment
Default
Security Check
Enable Security Check (/GS)
Enable Function-Level Linking
Yes (/Gy)
Enable Parallel Code Generation
Yes (/Qpar)
Enable Enhanced Instruction Set
Advanced Vector Extensions (/arch:AVX)
Floating Point Model
Precise (/fp:precise)
Enable Floating Point Exceptions
Create Hotpatchable Image
Enable String Pooling
Enables the compiler to create a single read-only copy of identical strings in the program image and in memory during execution, resulting in smaller programs, an optimization called string p...
OK
Cancel
Apply


MatrixRowVcc Property Pag
Configuration:
Active(Release)
Platform:
Active(x64)
Configuration Manager...
Common Properties
References
Configuration Properties
General
Debugging
VC++ Directories
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Heade
Output Files
Browse Information
Advanced
All Options
Command Line
Linker
All Options
/GS /Qpar /GL /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Fd"x64\Release\vc120.pdb" /fp:precise /D "_MBCS" /errorReport:prompt /WX- /Zc:forScope /arch:AVX /Gd /Oi /MD /Fa"x64\Release\" /EHsc /nologo /Fo"x64\Release\" /Fp"x64\Release\MatrixRowVcc.pch"
Additional Options
Inherit from parent or project defaults
/Qvec-report:2 /Qpar-report:2
OK
Cancel
Apply

Vectorizing Results (VC++)

loop	Vectorized	Reason
A	**Yes**	
B1/BV1	No/No	Outer loops
B2/BV2	No/No	Outer loops
B3/BV3	No/No	Non-contiguous accesses into an array/ Not enough type information
C1/CV1	No/No	Outer loops
C2/CV2	No/No	Outer loops
C3/CV3	No/No	Non-contiguous accesses into an array/ Not enough type information

Auto-Vectorization (GNU g++)

- The following compiler flag will enable the auto-vectorization
  - **-ftree-vectorize**
  - This flag is enable at **-O3**
- The following compiler flag will specify a targeted SIMD extension of IA32
  - **-mavx**
- The following compiler flag will report vectorizing results
  - **-ftree-vectorizer-verbose=2**
- Example
  - g++ -O3 -mavx -ftree-vectorizer-verbose=2 mytest.cpp

Vectorizing Results (GNU g++)

loop	Vectorized	Reason
A	**Yes**	
B1/BV1	NA/NA	
B2/BV2	**Yes**/No	Contains function call or data references that cannot be analyzed
B3/BV3	NA/No	Contains function call or data references that cannot be analyzed
C1/CV1	NA/NA	
C2/CV2	No/No	Complicated access pattern/Contains function call or data references that cannot be analyzed
C3/CV3	No/No	Unsupported use in stmt/Contains function call or data references that cannot be analyzed

- NA: No report is available

E. Parellelization and OpenMP

- Microsoft Visual C++ Compiler
  - /Qpar code generation option
  - /Qpar-report:2 additional option
  - **No loops are parallelized!**
- GNU g++ compiler
  - -floop-parallelize-all
  - -ftree-parallelize-loops=4
  - g++ 4.6.2 not implement the Graphite loop optimization
  - g++ 6.1.0 will compile, but no results are reported
- OpenMP directives
  - #pragma omp parallel for

OpenMP compiler flag

- -fopenmp

F. Performance Analysis

- The Dell Precision M4600 laptop is used for the testing
  - Processor (Intel(R) Core i7-2620M CPU) 2 cores and 4 threads
  - Windows 7 operating system
  - MInGW is installed for g++
- The auto-parallelization is tested only on the vector multiplication example with Visual C++.
  - The run-time with auto-parallelization is around 10 times slower than without parallelization.
- The run-times of vector multiplication (4 threads used)

	Auto-vector	AVX	OpenMP Auto-vector	OpenMP AVX
VC++	0.000050	0.000005	0.000005	0.000005
G++	0.000067	0.000001	0.000001	0.000001

AVX and OpenMP Using G++

loop	Vectorized	Reason
B1/BV1/BO1/BVO1	NA/NA/NA/NA	
B2/BV2/BO2/BVO2	**Yes**/No/NA/NA	Contains function call or data references that cannot be analyzed
B3/BV3/BO3/BVO3	NA/No/No/NA	Contains function call or data references that cannot be analyzed
C1/CV1/CO1/CVO1	NA/NA/NA/NA	
C2/CV2/CO2/CVO2	No/No/NA/NA	Complicated access pattern/Contains function call or data references that cannot be analyzed
C3/CV3/CO3/CVO3	No/No/No/NA	Unsupported use in stmt/Contains function call or data references that cannot be analyzed

- Four combinations: with/without AVX intrinsics and with/without OpenMP
- Auto-vectorization: Only row-major, without AVX intrinsics and OpenMP

The run-times of matrix multiplication (row-major)

	Normal (B)	AVX (BV)	OpenMP Normal (BO)	OpenMP AVX (BVO)
VC++ (4)	0.029821	0.004304	0.023691	0.004372
VC++ (2)	0.022238	0.005275	0.023987	0.004672
G++ (4)	0.026333	0.029333	0.066667	0.020333
G++ (2)	0.030667	0.030667	0.077667	0.025667

- The run-times are almost the same with 4 threads and 2 threads
- The run-times are also almost the same with and without OpenMP for VC++
- The run-time with OpenMP is much slower than without OpenMP for Normal for G++

The run-times of matrix multiplication (column-major)

	Normal (B)	AVX (BV)	OpenMP Normal (BO)	OpenMP AVX (BVO)
VC++ (4)	0.023848	0.004387	0.018558	0.004722
VC++ (2)	0.027695	0.004567	0.019088	0.004411
G++ (4)	0.093000	0.028333	0.055667	0.017667
G++ (2)	0.094000	0.026333	0.074333	0.020333

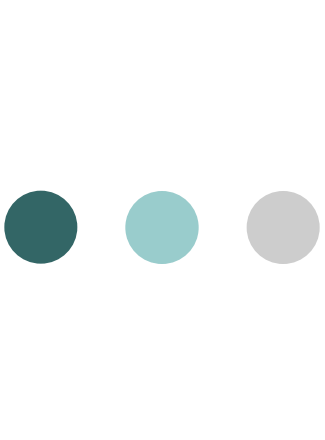
- The run-time for column-major is about 3 times slower than for row-major.
- The run-time is greatly reduced when AVX intrinsic functions are used for both VC++ and G++.

Conclusion

- Vectorization using intrinsics is necessary for non-trivial cases.
  - Auto-vectorization is performed for simple cases only.
  - g++, not vc++, can vectorize the inner-loop of matrix multiplication in row-major. But, both cannot vectorize the outer loop of matrix multiplication in column-major.
- Auto-parallelization is not as good as parallelization using Open-PM.
  - Using Open-MP directives to parallelize a loop is easy.
  - The run-time with auto-parallelization is around 10 times slower than without parallelization.

Conclusion (Cont.)

- The performance with vectorization is significant for both VC++ and G++ compilers.
- The performance with parallelization is not significant for both VC++ and G++ compilers.
  - It is sometimes worse!
  - May be because the testing laptop has only two cores and the size of matrixes is not large enough.
- Pick right algorithm and right data storage format is important too.
  - Row-major is 3-times faster than column-major when using g++.



Thank You

Questions?