CODE OPTIMISATION

ON INTEL XEON (CO)PROCESSORS

André Pereira ampereira@di.uminho.pt

Agenda

* The Case Study
* Identifying Inefficiencies
  * Common code pitfalls
  * Using compiler flags
* Vectorisation
* Shared Memory Parallelisation
* Intel Xeon Phi Coprocessor

The Case Study

```
void matMult (float **a, float **b, float **c, int size) {
  for (int i = 0; i < size; i++)
      for (int j = 0; j < size; j++) {
          c[i][j] = 0;
          for (int k = 0; k < size; k++)
              c[i][j] += a[i][k] * b[k][j];
      }
}
```

Common Code Inefficiencies

- Avoidable memory accesses
- False alias
- Blocking/Tiling
- Memory alignment - *to see later*
- Row/Column major - *work assignment…*

Avoidable Memory Accesses

```
void matMult (float **a, float **b, float **c, int size) {
	for (int i = 0; i < size; i++)
		for (int j = 0; j < size; j++) {
			c[i][j] = 0;
			for (int k = 0; k < size; k++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
}
```

Avoidable Memory Accesses

- Issue:
  - The same element in a data structure (matrix c) is being accessed twice from memory per cycle iteration
- Solution:
  - Use a temporary variable to store intermediate results

```
void matMult (float **a, float **b, float **c, int size) {
	for (int i = 0; i < size; i++)
		for (int j = 0; j < size; j++) {
			c[i][j] = 0;
			for (int k = 0; k < size; k++)
				c[i][j] = c[i][j] + a[i][k] * b[k][j];
		}
}
```

False Alias

- Issue:
  - a, b, or c may be pointing to overlapped memory blocks
  - Compiler is very cautions using optimisations
- Solution:
  - General C++ case prefer references over pointers!

```
void matMult (float **a, float **b, float **c, int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            c[i][j] = 0;
            for (int k = 0; k < size; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

False Alias

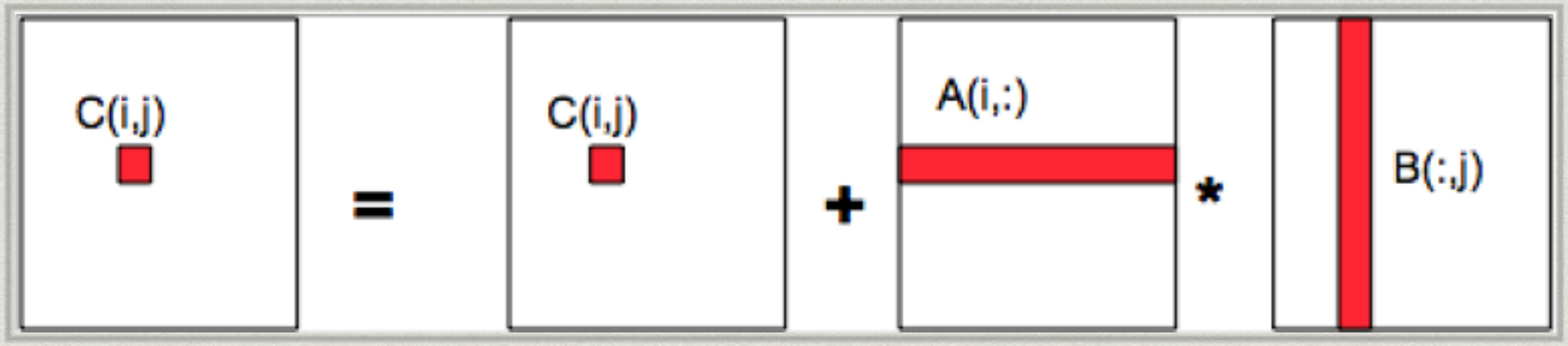
- Issue:
  - a, b, or c may be pointing to overlapped memory blocks
  - Compiler is very cautions using optimisations
- Solution:
  - General C++ case prefer references over pointers!
  - Give hints to the compiler (pragmas or restrict)

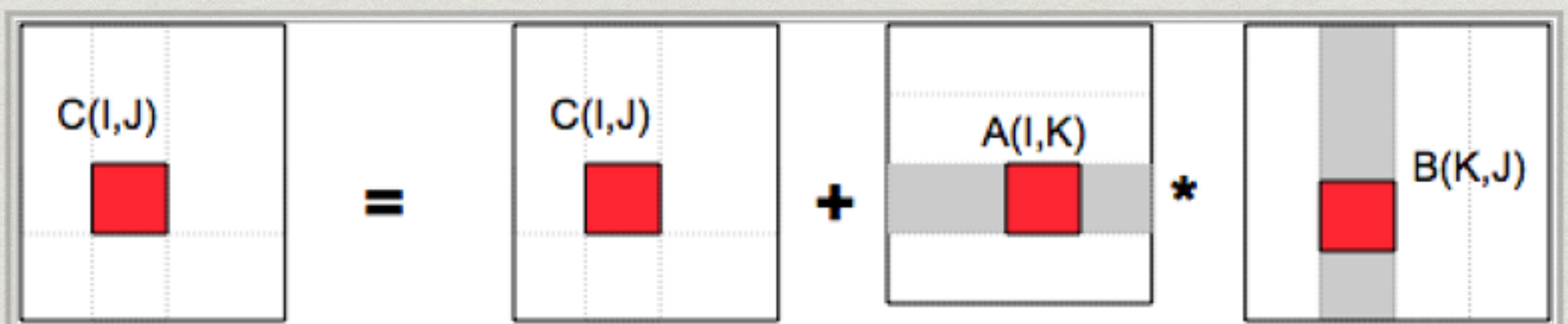
```
void matMult (float **a, float **b, float **c, int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            c[i][j] = 0;
            for (int k = 0; k < size; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Memory Alignment

- Issue:
  - Memory is not contiguously aligned
  - Alignment is not a multiple of 16 bytes
- Solution:
  - Guarantee the proper alignment of data structures manually

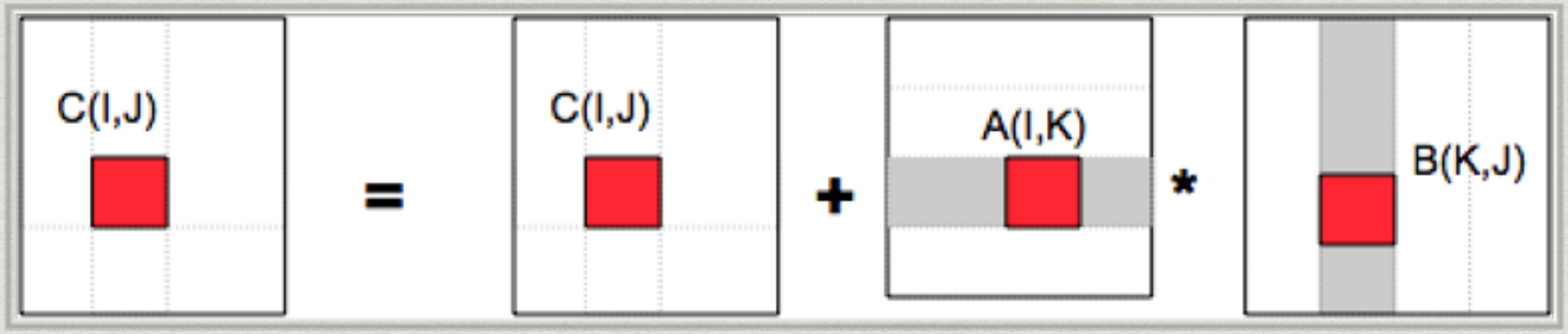
Blocking/Tiling





Blocking/Tiling

```
for I = 1 to N
      for J = 1 to N
       {read block C(I,J) into fast memory}
       for K = 1 to N
                 {read block A(I,K) into fast memory}
                 {read block B(K,J) into fast memory}
                  do a matrix multiply on blocks to compute block C(I,J)
        {write block C(I,J) back to slow memory}
```



Compiler Flags

- Compilers provide flags to enable sets of optimisations
- Both GNU and Intel compilers use the same syntax
- “Free lunch” speedups!

Compiler Flags

- -O1
  - Enable a small set of optimisations
  - Reduces both code size and execution time
  - Trade-off between performance and compilation time
- -O2
  - Performs almost all supported optimisations
  - Generates faster code without compromising the space
- -O3
  - Almost all available optimisations without considering any trade-off

Vectorisation

- How is it achieved? By hand?

```
void matrixAdd (void) {
	__m256 ymm1, ymm2;

	for (unsigned i = 0; i < SIZE; ++i) {
		for (unsigned j = 0; j < VEC_SIZE; ++j) {
			ymm1 = _mm256_load_ps(&m1[i][j * 8]);
			ymm2 = _mm256_load_ps(&m2[i][j * 8]);

			ymm1 = _mm256_add_ps(ymm1, ymm2);
			_mm256_store_ps(&result[i][j * 8], ymm1);
		}
	}
}
```

Vectorisation

- Loop vectorisation is enabled by default with -O3 for the GNU compiler and -O2 for Intel compiler
- ...but is the code really vectorised?

```
src/matrix.cpp(102): (col. 3) remark: PERMUTED LOOP WAS VECTORIZED.
src/matrix.cpp(102): (col. 3) remark: REMAINDER LOOP WAS VECTORIZED.
src/matrix.cpp(105): (col. 4) remark: loop was not vectorized: not inner loop.
src/matrix.cpp(101): (col. 2) remark: loop was not vectorized: not inner loop.
```

```
void matMult (float **a, float **b, float **c, int size) {
    for (int i = 0; i < size; i++)
        for (int j = 0; j < size; j++) {
            c[i][j] = 0;
            for (int k = 0; k < size; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}
```

Vectorisation

- Loop vectorisation is enabled by default with -O3 for the GNU compiler and -O2 for Intel compiler
- ...but is the code really vectorised?

```
src/matrix.cpp(102): (col. 3) remark: PERMUTED LOOP WAS VECTORIZED.
src/matrix.cpp(102): (col. 3) remark: REMAINDER LOOP WAS VECTORIZED.
src/matrix.cpp(105): (col. 4) remark: loop was not vectorized: not inner loop.
src/matrix.cpp(101): (col. 2) remark: loop was not vectorized: not inner loop.
```

```
void matMult (float **a, float **b, float **c, int size) {
```

```
src/matrix.cpp(244): (col. 2) remark: loop was not vectorized: existence of vector dependence.
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between  line 244 and  line 244.
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between  line 244 and  line 244.
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between  line 244 and  line 244.
src/matrix.cpp(244): (col. 2) remark: vector dependence: assumed OUTPUT dependence between  line 244 and  line 244.
```

```
) {
size; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
}
```

Vectorisation

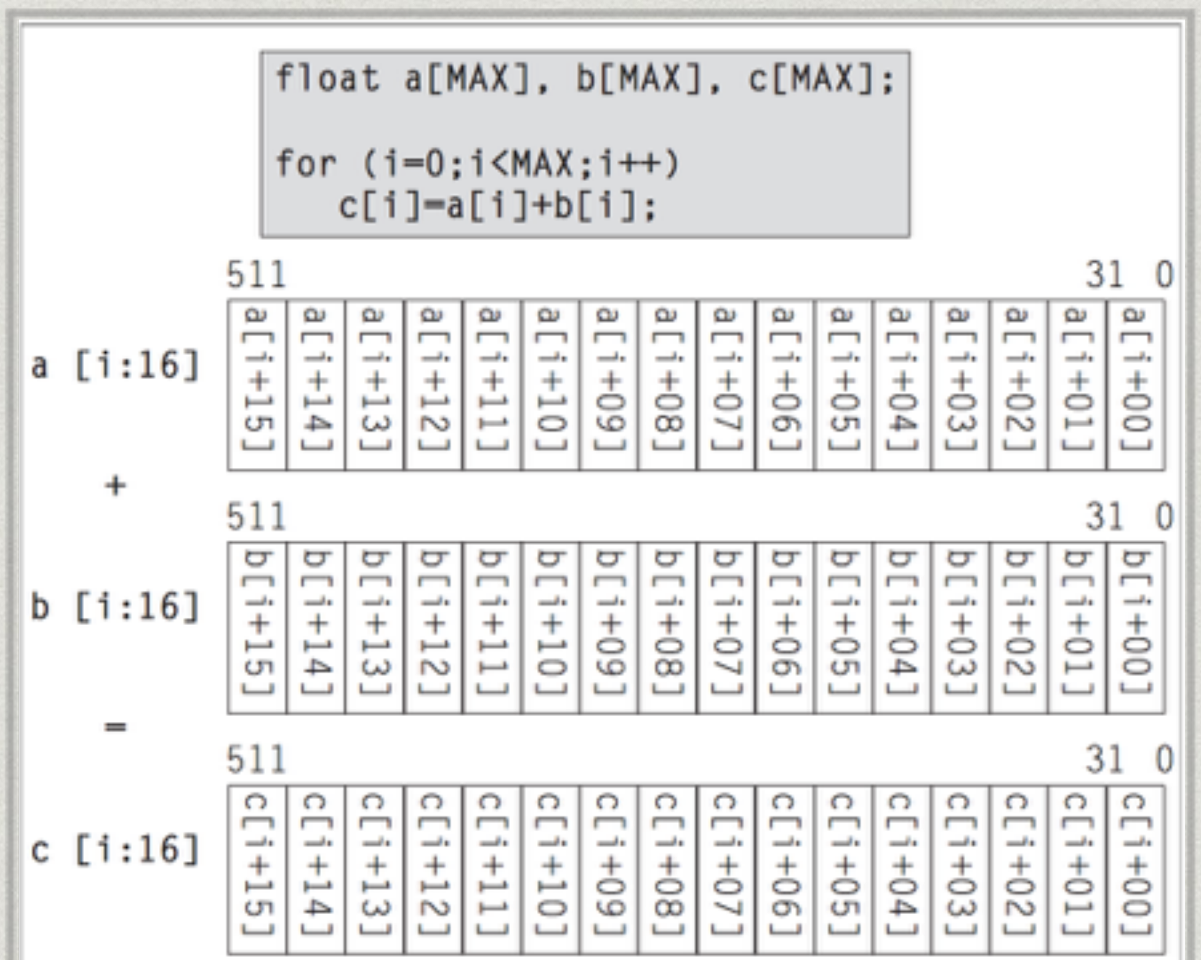
- Vectorisation report ICC: -vec-reportX, where X
  - 1 - Loops successfully vectorised
  - 2 - Loops not vectorised (and the justification)
  - 3 - Adds dependency information
  - 4 - Non-vectorised loops report
  - 5 - Non-vectorised loops report with dependency information

Help the compiler vectorise

- Give information about loop dependencies
  - #pragma vector always
  - #pragma ivdep
- Avoid nested loops
  - or use #pragma omp simd collapse(X) - OpenMP 4.0
- Data alignment and layout

Data Layout

- Ensure that vector operations are applied to packed data; Ex. 512-bit Xeon Phi registers



Data Alignment

- Ensure that the memory allocation of static data structures is properly aligned
- Depends on the size of the vector registers
    - `__attribute__((align(DIM_SIMD))) float a [SIZE];`

EXERCISES - PART I

Exercise 1

- Perform basic code optimisations
- Parallelise code execution for shared memory systems
- https://bitbucket.org/ampereira/matrix-optimization/downloads

Shared Memory Parallelism

- Several libraries to produce parallel code for shared memory environments
  - **OpenMP**
  - TBB
  - CILK
  - …

OpenMP

- Easy to use, pragma-based
- Implemented by default in both GNU and Intel compilers
  - -fopenmp - gcc
  - -openmp - icc
- Add the <omp.h> header to the code and use the pragmas

(Very) Basic Pragmas

- #pragma omp parallel options

  #pragma omp for/task

- Options:
  - num_threads
  - schedule(X)
  - private(X)

Go In-depth

- Several work scheduling heuristics available
- Definition of the task grain
- Data scoping
- Reduction, synchronisation, nowait clauses…
- Parallel tasks for irregular problems
- Nested parallelism

EXERCISES - PART II

Using Intel Composer on SeARCH

- Always have 2 terminal windows open
  - The computing node (interactive *qsub*)
  - The frontend, where the code is compiled
- Run *module load intel/2013.1.117* on both windows
  - In the computing node to set the dynamic library paths
  - In the frontend to use the compiler, headers, etc
- If you get "Catastrophic error: could not set locale "" to allow processing of multibyte characters" do *export LC_ALL=C*

References

- An Overview of Programming for Intel® Xeon® processors and Intel® Xeon Phi™ coprocessors, James Reinders, Intel
- Intel® Xeon Phi™ Coprocessor High Performance Programming, Jim Jeffers, James Reinders, Elsevier Waltham (Mass.), 2013
- Intel® 64 and IA-32 Architectures Software Developer's Manual