FINM 32950: Intro to HPC in Finance

Lecture 2

Chanaka Liyanarachchi

March 31, 2025

Vectorization

Using Multicore Nodes

Vectorization

Vectorization in Action

- Last week: we introduced the idea of vectorization.
- Today: we look at how vectorization works using examples.
- We will look at 2 ways to introduce vectorization:
  1. Using intrinsics (low-level functions built in to the compiler)[1].
  2. Auto vectorization (using high-level language constructs).
- Intrinsics is a powerful technique but we do not use it directly in this course. We use intrinsics mainly to demonstrate/prove vectorization in action.
- In this course we prefer auto vectorization (for reasons we shall discuss later).

[1] `https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=vs-2019`

Representing Packed Data

- C++ defines data types for fundamental types.
- E.g.: int, double, float, short
- We cannot directly use them to store packed data.
- We need to use *vector extension types* to store packed data.
- The register size and fundamental data type determine how many data items we can store in a packed-register:
  - 128 bit register: can store 4 floats or 2 doubles
  - 256 bit register: can store 8 floats or 4 doubles
- Vector extension (packed data) types depend on:
  1. Size of the registers (register size depends on the architecture, SSE2: 128; AVX2: 256; AVX-512: 512)
  2. Fundamental data type (individual data item size depends on type, float: 32; double:64)

Packed Data Types

- For 128 bit registers:
  - `__m128`: stores 4 floats
  - `__m128d`: stores 2 doubles
  - `__m128i`: stores 4 ints
  - ...
- For 256 bit registers:
  - `__m256`: stores 8 floats
  - `__m256d`: stores 4 doubles
  - `__m256i`: stores 8 ints
  - ...
- There's a pattern:
  - prefix: `__m`
  - size of the register (e.g. 128, 256)
  - fundamental data type (i: int; d: double; default: float; ...)

Operations on Packed Data

- We must use intrinsic functions (packed instructions) to operate on packed data (i.e., we cannot use regular arithmetic operators (+, -, etc.) with packed types).
- Intrinsics provide a *family of functions* for a given operation (e.g. addition) for each packed type.
- Example: for SSE2:
  - `_mm128_add_epi32`: to add packed (4) integer values stored in a 128 bit packed register
  - `_mm128_add_epi16`: to add packed (8) short values stored in a 128 bit packed register
  - ...
- Example: for AVX2:
  - `_mm256_add_epi32`: to add packed (8) integer values stored in a 256 bit packed register
  - `_mm256_add_epi16`: to add packed (16) integer values stored in a 256 bit packed register
  - ...
- Ref: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/#=undefined`

Intrinsics: Example 1

- Example addition for SSE processors:

```
#include <xmmintrin.h> // for SSE
#include <stdio.h> //for printf

void add_test()
{
    __m128 a = _mm_set_ps(1.0f, 2.0f, 3.0f, 4.0f);
    __m128 b = _mm_set_ps(5.0f, 6.0f, 7.0f, 8.0f);

    __m128 c = _mm_add_ps(a, b);

    // displaying results
    float* f = (float*)&c;
    printf("%f %f %f %f\n", f[0], f[1], f[2], f[3]);
}
```

Intrinsics: Example 2

- Let's look at another example to show the power of vectorization.
- Suppose we want to do a multiply and add operation on vectors:

```
for (int i=0; i<16; i++)
{
   d[i] = a[i] * b[i] + c[i];
}
```

- This for loop uses 16 additions and 16 multiplications.
- We can do all of them in one operation:

```
__m512 a = /*packed data*/
__m512 b = /*packed data*/
__m512 c = /*packed data*/

__m512 d = _mm512_fmadd_ps(a, b, c);
```

Vectorization using Intrinsics: Advantages and Disadvantages

- We briefly looked at intrinsics to show:
  1. How vectorization works
  2. Benefits/power of vectorization
- Intrinsics has some drawbacks:
  - Code is difficult to read, write, and maintain (try to write a Black Scholes pricer)
  - Code is not portable
- Next, we will look at how to achieve vectorization using high-level programming constructs.

Auto-Vectorization

Auto-Vectorization

- Modern compilers can vectorize code *automatically*. It is known as auto-vectorization.
  - The compiler generates packed SIMD instructions to replace loops.
- For auto-vectorization to work:
  - Code has to satisfy some requirements
  - Certain directives have to be used to encourage/force the compiler auto-vectorize.
- We need to understand how we can write code to maximize auto-vectorization:
  - If auto-vectorization occurred
  - What if it did not? we need to know why the compiler did not vectorize the loop/loops
  - So, we can do something to *encourage/force* compiler to auto-vectorize

Vectorization Using Intel C++ Compiler

- Modern compilers support auto vectorization, but details are vendor specific.
- We will limit our discussion to the **Intel C++ compiler** (works on Windows, Linux and Mac).
- General ideas about vectorization however are not vendor specific.

Auto-Vectorization: Example 1

- Let's add two vectors:

```
int main()
{
    const int N = 8;
    int a[N] = {1, 2, 3, 4, 5, 6, 7, 8};
    int b[N] = {1, 2, 3, 4, 5, 6, 7, 8};
    int c[N];

    for (int i = 0; i < N; ++i)
        c[i] = a[i] + b[i];
}
```

- Intel compiler supports following optimizations:

```
-O1       optimize for maximum speed, but disable some optimizations which
          increase code size for a small speed benefit
-O2       optimize for maximum speed (DEFAULT)
-O3       optimize for maximum speed and enable more aggressive optimizations
          that may not improve performance on some programs
-O        same as -O2
-Os       enable speed optimizations, but disable some optimizations which
          increase code size for small speed benefit
-O0       disable optimizations
-Ofast    enable -O3 -no-prec-div -fp-model fast=2 optimizations
```

- Use help for all available options:

```
icc -help
```

- Compiler is expected to auto-vectorize if optimization level 2 (O2) or higher is used.
- Will not vectorize if O1 or lower is used.
- Using O2:

```
icc -O2  example1.cpp -o example1
```

- Is the loop above auto vectorized?

Vectorization Report

- A *vectorization-report* shows what loops were vectorized and explains why any loop was not vectorized. To generate a vectorization report:
- On Linux:

  ```
  -qopt-report[=n]

  Generates an optimization report. Default destination is
  <target>.optrpt.  Levels of 0 - 5 are valid.
  ```

Vectorization Report

- Each level provides different amount of of information:
  - n=1: Loops successfully vectorized
  - n=2: Loops not vectorized; and the reason why not
  - n=3: Adds dependency information
  - n=4: Reports only non-vectorized loops
  - n=5: Reports only non-vectorized loops and adds dependency info
- The report file has the .REP extension on Windows and .optrpt on Linux.
- Additionally, we use the following to limit the output to vectorization only as the reports are generally lengthy and difficult to understand at first (you should experiment with and without this flag):

```
-qopt-report-phase=vec
```

- Let's create a vectorization report:
  `icc -qopt-report=1 -O2  example1.cpp -o example1`
- In the report (example1.optrpt) we see that this loop is vectorized:

```
LOOP BEGIN at example1.cpp(9,2)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- Let's build the same program with optimizations disabled:
  `icc -qopt-report=1 -O0  example1.cpp -o example1`
- The loop is not vectorized.
- Works as advertised.

Auto-Vectorization: Example 2

- Here's another example[2]:

```
int main()
{
   int sum = 0;

   for (int row = 0; row < ROWS; ++row)
   {
      for (int col = 0; col < COLS; ++col)
      {
         data[row][col] = row + col;
      }
   }

   for (int row = 0; row < ROWS; ++row)
   {
      for (int col = 0; col < COLS; ++col)
      {
        sum += data[row][col] + data[col][row];
      }
   }
}
```

[2] This example uses some arbitrary operations on a matrix, just for illustration.

- We have several (for) loops now.

  `icc -qopt-report=1 -O2  example2.cpp -o example2`

- First inner loop is vectorized, but the second inner loop is not vectorized.

```
LOOP BEGIN at example2.cpp(13,7)
   remark #15300: LOOP WAS VECTORIZED
LOOP END


LOOP BEGIN at example2.cpp(21,7)
   remark #15335: loop was not vectorized:
LOOP END
```

Intel Advisor

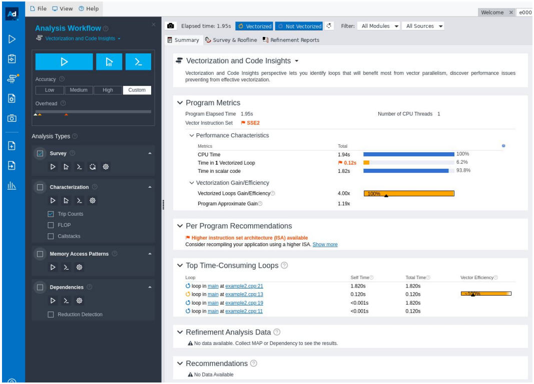
- Intel toolkit has many features to write performance sensitive code.
- Intel Advisor helps identify under optimized loops related info.
- Let's use Advisor to analyse Example 2 above.

Using Intel Advisor on Midway

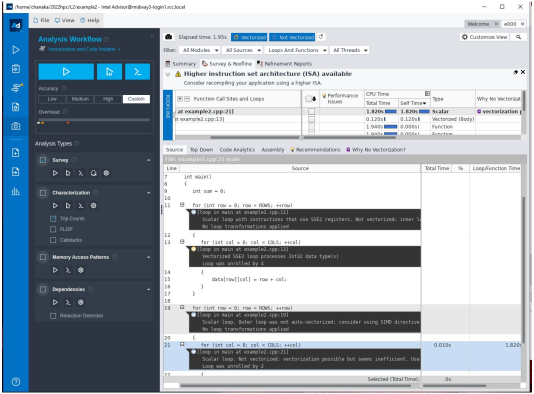
- Load the modules:
  ```
  module use /software/intel/oneapi_hpc_2022.1/modulefiles

  module load advisor/2022.0.0
  ```
- Copy L2Demo.tar from /projects/finm32950/chanaka/ directory and untar (uncompress it).
- Change directory to L2Demo/advisor directory:
  ```
  cd L2Demo/advisor
  ```
- Build:
  ```
  make
  ```
- Collect survey data:
  ```
  advisor --collect=survey --project-dir=.  ./example2
  ```
- Generate survey report:
  ```
  advisor --report=survey --project-dir=.
  ```
- Examine report:
  ```
  advisor-gui ./
  ```
- Use "Show Results" to see the results.

- Use summary tab to view the summary report:



- Use survey and roofline tab to get details:



Barriers to Vectorization

Example 3

- Let's look at this example:

```
#include <cstdlib>

int main()
{
   const int N = 8;
   float a[N], b[N], c[N];

   for (int i=0; i<N; ++i)
   {
      a[i] = rand() % 100;
      b[i] = rand() % 100;
   }

   for (int i = 0; i < N; ++i)
      c[i] = a[i] + b[i];
}
```

- We have 2 for loops. Will the compiler vectorize them?

- Let's compile this (O2 is default; we don't need to type it):
  `icc -qopt-report=1 example3.cpp`
- Vectorization report:

```
LOOP BEGIN at example3.cpp(8,2)
   remark #25436: completely unrolled by 8
LOOP END

LOOP BEGIN at example3.cpp(14,2)
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- Let's compile this again to find out why the first loop is not vectorized:

  ```
  icc -qopt-report=2 example3.cpp
  ```

- Vectorization report:

  ```
  LOOP BEGIN at example3.cpp(8,2)
     remark #15344: loop was not vectorized: vector
     dependence prevents vectorization.
     First dependence is shown below. Use level 5 report
     for details

  LOOP BEGIN at example3.cpp(14,2)
     remark #15300: LOOP WAS VECTORIZED
  LOOP END
  ```

- Now, the vectorization-report to guides us (and, compiling this again):

  `icc -qopt-report=5 example3.cpp`

- Vectorization report:

```
LOOP BEGIN at example3.cpp(8,2)
   remark #15382: vectorization support: call to function
   rand() cannot be vectorized   [ example3.cpp(10,10) ]
```

- This loop is not vectorized because we are using `rand()` which doesn't support vectorization.

Example 4

- Another example of a loop that's not auto-vectorized:

  ```
  for (int i = 0; i < N; ++i)
     if(i>3) c[i] = a[i] + b[i] + c[i-1];
  ```

- This loop won't be auto vectorized:

  ```
  remark #15344: loop was not vectorized: vector dependence
  prevents vectorization.
  ```

- Why?

- Let's unroll this loop to see what's going on:

```
c[0] = a[0] + b[0];
c[1] = a[1] + b[1];
.....
c[4] = a[4] + b[4] + c[3];
c[5] = a[5] + b[5] + c[4];
.....
```

- We cannot calculate `c[4]` and beyond in parallel due to dependancy to other elements.
- This data dependance prevents vectorization.

Example 5

- Let's look at this example, which is similar to Black Scholes pricer where we are passing several arrays to a function:

```
void add(float *a, float *b, float *c,
   float *d, float *e, int N)
{
   for (int i=0; i<n; ++i)
      a[i] = b[i] + c[i] + d[i] + e[i];

}
```

- Having many pointers may make it difficult for the compiler to figure out any dependence.
- Compiler may not know if the pointers are pointing to aliased memory.
- We have a data dependency problem if elements are overlapped.

- You may see a vectorization report similar to what's shown below, where one entry indicates the loop is vectorized:

```
LOOP BEGIN at example4.cpp(7,5)
<Multiversioned v1>
   remark #25228: Loop multiversioned for Data Dependence
   remark #15300: LOOP WAS VECTORIZED
LOOP END
```

- And, another entry indicates the same loop is not vectorized:

```
LOOP BEGIN at example4.cpp(7,5)
<Multiversioned v2>
   remark #25436: completely unrolled by 8
LOOP END
```

- We have 2 versions for the same loop. What's going on?

Multiversions and Unrolling Loops

- Compiler may generate two versions (a vectorized version and a non-vectorized version) when it is not sure if vectorization is safe.
- Sometimes a compiler may unroll a loop instead of vectorizing it.
- Run time uses most appropriate one (i.e. vectorization is not guarenteed) so this is not ideal.

Example 6

- Loop count should remain constant during the execution of the loop for auto vectorization.

  ```
  while (i<N)
  {
     a[i] = a[i] * b[i];

     if (a[i] < 2) break;

     ++i;
  }

  remark #15520: loop was not vectorized: loop with multiple
  exits cannot be vectorized
  ```

- A loop must have a single entry and a single exit (entire loop must run) for auto vectorization.

Reading Vectorization Reports

- Examples above show some (not all) common vectorization reports.
- We can use them to learn how read and understand vectorization reports.
- You may see similar types of reports for other applications.

Improving Vectorization

- We saw some cases where auto vectorization is not used/possible.
- Shown below are some techniques to improve vectorization in such cases:
    1. Using vectorized implementations (e.g. SVML)
    2. Compiler directives
    3. Writing code to support vectorization

1. Using vectorized implementations - Short Vector Math Library (SVML)

- A loop is not vectorized if a non vectorizable function is used in a loop.
- One solution is to use vectorized implementations of such functions.
- *Intel Short Vector Math Library* library provides vectorized implementations for some functions:

```
void test(float* a, float* b, int n)
{
   for (int i=0; i<n; ++i)
   {
      a[i] = sinf(b[i]);
   }
}
```

- A loop that uses vectorized functions will be auto vectorized.

- Support for vectorized implementations in the SVML include: sin, cos, tan, asin, acos, atan, log, log2, log10, exp, exp2, sinh, cosh, tanh, asinh, acosh, atanh, erf, erfc, erfinv, sqrt, cbrt, trunc, round, ceil, floor, fabs, fmin, fmax, pow, atan2
- Reference: `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/intrinsics-for-short-vector-math-library-ops.html`

2. Compiler Directives

- The compiler uses a static analysis to determine if vectorization safe (i.e produce correct results).
  - Won't vectorize any code unless it can guarentee correctness.
  - May ignore vectorization even when vectorization is possible due to ambiguity.
- Compilers provides directives so the programmer can guide/force vectorization when we (the programmer) know it is safe to vectorize such code.
- A guidance usually applies to a localized area (e.g. a function, a loop).

- The Intel C++ supports following pragma directives to help vectorization:
  - simd: instructs the compiler to enforce vectorization
  - ivdep: instructs the compiler to ignore assumed vector dependencies (guidance to the compiler)
  - vector always: force vectorization when "vectorization possible but seems inefficient"
- Related:
  - novector: tells the compiler not to vectorize a loop
  - unroll: unroll a loop
  - nounroll: don't unroll a loop
- Ref: `https://www.intel.com/content/www/us/en/docs/cpp-compiler/developer-guide-reference/2021-8/pragmas.html`
- We will discuss openmp directives next week.

Example 4 (cont..): simd Keyword

- Compiler generated two versions in this case, due to pointer aliasing (Example 4). We can force vectorization if we know pointer aliasing is not an issue in our application:

```
void add(float *a, float *b, float *c,
        float *d, float *e, int n)
{
   #pragma simd
   for (int i=0; i<n; i++)
     a[i] = b[i] + c[i] + d[i] + e[i];
}
```

Example 4 (cont..): ivdep Keyword

- Or, we could use:

```
void add(float *a, float *b, float *c,
        float *d, float *e, int n)
{
   #pragma ivdep
   for (int i=0; i<n; i++)
     a[i] = b[i] + c[i] + d[i] + e[i];
}
```

Example 2 (cont..): vector Keyword

- Compiler generated two versions for the second inner loop earlier (Example 2):

```
for (int row = 0; row < ROWS; ++row)
{
      for (int col = 0; col < COLS; ++col)
      {
          data[row][col] = row + col;
      }
}

for (int row = 0; row < ROWS; ++row)
{
      #pragma vector always
      for (int col = 0; col < COLS; ++col)
      {
         sum += data[row][col] + data[col][row];
      }
 }
```

- Both inner loops are vectorized now.

3. Writing Code to Support Vectorization

- Following guidelines may improve auto vectorization:
  1. Simplify loops. Avoid multiple exits and complex loop termination conditions.
  2. Avoid data dependance in loops:
     - E.g.: Move the first and/or the last iterations out of the loop to remove dependancies (known as loop peeling).

       Before

       ```
       for (int i=0; i<10; ++i}
       {
          a[i] = b[i] + a[0];
       }
       ```

       After

       ```
       a[0] = b[0] + a[0];
       for (int i=1; i<10; ++i)
       {
           a[i] = b[i] + a[0];
       }
       ```

  3. Avoid mixing types in the same loop.
  4. ...

Unrolling Loops

- We cannot vectorize every loop. Unrolling can be helpful when vectorization is not possible.
- Loops is an important control structure that allow us to write concise and readable code.
- Unrolling is a technique used to optimize a loop by minimizing the cost of loop overhead:
  - Evaluate termination condition.
  - Update loop counter/counters.
- Unroll directives allow us to write code using loops but unroll them at compile time.

- Suppose we have a loop:

```
for (int i=0; i<4; ++i)
{
    a[i] = b[i] + c[i];
}
```

- Fully unrolling it:

```
a[0] = b[0] + c[0];
a[1] = b[1] + c[1];
a[2] = b[2] + c[2];
a[3] = b[3] + c[3];
```

- Unrolling by a factor of two:

```
for (int i=0; i<4; i+=2)
{
    a[i] = b[i] + c[i];
    a[i+1] = b[i+1] + c[i+1];
}
```

- Code is executed sequentially – not as efficient as vectorization.
- Performance improvements depend on the unrolling factor and the best factor can only be determined through measurements (discussed last week).

Vectorization: Usage

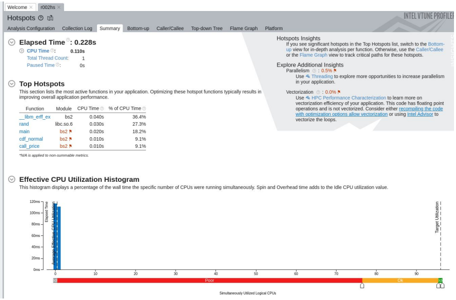
- Many popular libraries use vectorization to optimize operations:
  1. Eigen: `http://eigen.tuxfamily.org/index.php?title=Main_Page`
  2. RapidJSON: `https://github.com/Tencent/rapidjson`
  3. NumPy: `https://www.pythonlikeyoumeanit.com/Module3_IntroducingNumpy/VectorizedOperations.html#`

Remarks

- Vectorization is important.
- We discussed 2 ways to introduce vectorization:
  1. Using processor intrinsic functions
  2. Compiler auto vectorization
- Every application is different. There are no hard and fast rules that guarantee auto-vectorization.
- Recommendations/techniques described above may increases the success rate.

Using Multicore Nodes

- Let's go back to the report from vtune last time and focus on the *Effective CPU Utilization Histogram* section.



- We have over 90 CPU cores but we just use one.
- How do we use more than one CPU cores?

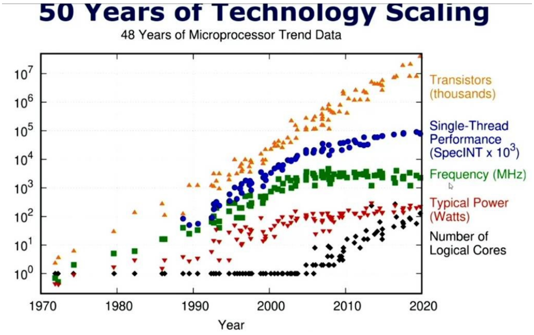
Parallel Programming - Introduction

Parallel Programming

- One obvious way get things done faster is to do more than one task simultaneously (in parallel):
  - This idea has been around for a long time.
  - It has become a very important topic now.
- We've seen vectorization parallelism already, and it works within one CPU core, giving us parallelism at the instruction level.
- Our next topic is parallelism using multithreading, allowing us to use many CPU cores.

Moore's Law

- Let's first try to understand why parallel programming using multi cores is needed.
- Moore's law says: every two years computing power doubles.
- Graphs below confirm that, but only upto about 2010. [3]



[3] source: `https://www.datacenterknowledge.com/supercomputers/after-moore-s-law-how-will-we-know-how-much-faster-computers-can-go`

- Frequency has hit a limit due to hardware limitations (due to leakage currents in transistors).
- Single thread performance is still improving mainly due to vectorization.
- Trend now is to have more and more CPU cores.
  - Achieve same performance or better.
  - Less power consumption and heat generation.

Learning Parallel Programming

- New challenge for us (programmers):
  - We should know how to use multicores.
  - Otherwise, we are wasting a lot of computing power.
  - Learning how to write sequential code is not enough anymore.
- Parallel computing is not a highly-specialized/exotic topic anymore; it is something every programmer should know.

Parallel Computing: Example 1

- Suppose we want to add two arrays (vectors):

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} + \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \end{bmatrix}$$

- We can write a simple for loop:

```
for (int i=0; i<N; ++i)
    c[i] = a[i] + b[i];
```

- This loop adds two values and assign it to $c[i]$ for each i in N.
- Usually when we think about programs (thus far), we think about doing things sequentially:
  - Use one data item from each array
  - Executing one instruction at a time
- Each addition is independent of the others (i.e. `c[0]` doesn't depend on `c[1]` and so on) – we can compute them in parallel.

Parallel Computing: Example 2

- Matrix multiplication is another example:

$$A_{I,K}B_{K,J} = C_{I,J} \qquad c_{i,j} = \sum_{k=0}^{K-1} a_{i,k}b_{k,j} \text{ where } 0 \le i < I \text{ and } 0 \le j < J$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,k-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i-1,0} & a_{i-1,1} & \cdots & a_{i-1,k-1} \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,j-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,j-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k-1,1} & b_{k-1,2} & \cdots & b_{k-1,j-1} \end{pmatrix}$$

$$= \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,j-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,j-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{i-1,0} & c_{i-1,1} & \cdots & c_{i-1,j-1} \end{pmatrix}$$

- Each element $c_{i,j}$ is independent of others – we can compute them in parallel.

Natural Parallelism

- Many applications in real life exhibit natural parallelism - vector addition and matrix multiplication are two examples.
- Our goal in this course is to identify and use natural parallelism to speed up program executions.
- We will explore 3 *technologies:*
  1. Vectorization ✓
  2. Multicore
  3. GPGPU
- FPGA is another interesting technology but we won't have time to discuss that topic.

Multithreading

Threads

- To use more than one core (multi cores), we break up a program into smaller *tasks*.
- We use *threads* to execute these tasks on different cores.
- This is generally known as multithreading.
- Our next topic is to explore how to use threads to do work in parallel.
- First, let's look at some important concepts about using threads.

Using Threads: Example 1

- Let's extend our familiar *Hello World* example to use a thread to perform a task.
- Task in this case is to write the greeting message to console.

```
#include <iostream>
#include <thread>

void Greeting()
{
   std::cout << "Hello, World" << std::endl;
}

int main()
{
   std::thread t(Greeting);

   t.join();
}
```

To build, on midway:

```
icc helloworld.cpp -o helloworld -lpthread
```

Hello World Example

- To create a thread, we use `std::thread`.
- The `std::thread` constructor takes a function as an argument.
- Thread execution begins in this function.
- Once the new thread is started, we have two threads in the program:
  1. main thread
  2. new thread (t)
- We use `join()` to make sure the calling thread (main) waits until new thread (t) finishes its execution.
- Thread is defined in `<thread>`
  `http://www.cplusplus.com/reference/thread/thread/`

Thread Join and Detach

- Once a thread is created:
  - calling thread waits until the new thread completes its execution: use `join()`
  - calling thread continues execution without waiting for the new thread: use `detach()`

How Many Threads

- We can create a large number of threads in a program.
- How many threads can run in parallel depends on the number of cores/cpus.
- If we create a large number of threads, every thread may not run in parallel.
- The OS uses a scheduling algorithm to run the threads.

Detour: Lambdas (C++)

Lambdas

- We saw the `std::thread` constructor takes a function as an arugment.
- There are other ways to pass an initial function to a thread:
  - using a lambda
  - using a function object

Lambda: Syntax

- The example below shows a very simple lambda which writes a message to console:

```
[] (string s)
{
  cout << s << endl;
}
```

- [] is called the capture clause; a lambda is always introduced by the capture clause.
- () parenthesis allows us to pass optional arguments to a lambda.
- {} lambda body is enclosed by the {} brackets.

Creating a Thread

- We can pass a lambda to a thread:

```
std::thread t([]()
{
        cout << "Hello, World" << endl;
});

t.join();
```

Lambda Syntax: Captures

- Lambdas allow us to *capture* parameters by value, or by reference.
- You can use *everything* notation to capture. In this case the compiler captures what's needed (used inside the body) by the lambda.
- `[=]` captures everything by value – allows reads but no write access.
- `[&]` captures everything by reference – allows read and write access.
- `[=,&x]` captures everything by value except `x`; `x` is captured by reference.
- `[&, x]` captures everything by reference except `x`; `x` is captured by value.

Using Threads: Example 2

- Matrix multiplication:

$$A_{I,K} B_{K,J} = C_{I,J} \qquad c_{i,j} = \sum_{k=0}^{K-1} a_{i,k} b_{k,j} \text{ where } 0 \le i < I \text{ and } 0 \le j < J$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,k-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,k-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i-1,0} & a_{i-1,1} & \cdots & a_{i-1,k-1} \end{pmatrix} \begin{pmatrix} b_{0,0} & b_{0,1} & \cdots & b_{0,j-1} \\ b_{1,0} & b_{1,1} & \cdots & b_{1,j-1} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k-1,1} & b_{k-1,2} & \cdots & b_{k-1,j-1} \end{pmatrix}$$

$$= \begin{pmatrix} c_{0,0} & c_{0,1} & \cdots & c_{0,j-1} \\ c_{1,0} & c_{1,1} & \cdots & c_{1,j-1} \\ \vdots & \vdots & \ddots & \vdots \\ c_{i-1,0} & c_{i-1,1} & \cdots & c_{i-1,j-1} \end{pmatrix}$$

- We can write a serial program:

```
void matrix_multiply(const matrix& m1, const matrix& m2,
   matrix& m3, int rows, int columns)
{
   for (int i = 0; i < rows; ++i)
   {
      for (int j = 0; j < columns; ++j)
      {
            m3[i][j] = 0;
            for (int k = 0; k < rows; ++k)
            {
                m3[i][j] += m1[i][k]*m2[k][j];
            }
        }
    }
}
```

- Inputs (m1 and m2) are passed by const reference.
- Output (m3) passed by reference.

- This is a naturally parallel program - each $c_{i,j}$ can be computed in parallel.
- We can parallelize it in many different ways:
  - one thread to calculate each element $c_{i,j}$ – we need numRow * numColumn number of threads
  - one thread for one row – we need numRow number of threads
  - one thread for one column – we need numColumn number of threads
- Which choice is better?

- Suppose we use one thread for each element:

```
void multiply_matrix(const matrix& m1, const matrix& m2,
    matrix& m3, int rows, int columns)
{
    vector<thread> threads(rows*columns);

    for (int i = 0; i < rows; ++i)
    {
        for (int j = 0; j < columns; ++j)
        {
            int idx = i*columns + j;
            threads[idx] = thread([=,&m3]()
            {
                m3[i][j] = 0;
                for (int k = 0; k < rows; ++k)
                {
                    m3[i][j] += m1[i][k] * m2[k][j];
                }
            });
        }
    }

    for (thread& t : threads)
    {
        t.join();
    }
```

- Or, we can to use one thread for one row:

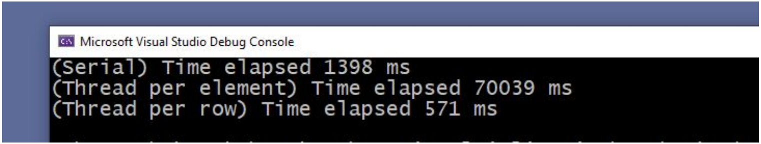
```
void multiply_matrix(const matrix& m1, const matrix& m2,
    matrix& m3, int rows, int columns)
{
    vector<thread> threads(rows);

    for (int i = 0; i < rows; ++i)
    {
        threads[i] = thread([=,&m3]()
        {
            for (int j = 0; j < columns; ++j)
            {
                m3[i][j] = 0;
                for (int k = 0; k < rows; ++k)
                {
                    m3[i][j] += m1[i][k] * m2[k][j];
                }
            }
        });
    }

    for (thread& t : threads)
    {
        t.join();
    }
```

- Note: This program does not use any special features of Intel compiler (i.e. you may use Visual C++ or any other compiler).
- We have 3 implementations now:
  1. sequential work – one thread (i.e. main thread) computes all elements
  2. one thread computes one element
  3. one thread computes one row
- Which program/programs will run faster?

- Let's run these programs and measure time.
- Results depend on the system and the matrix sizes etc.
- Shown below is what I saw for one case (see demo for code) on my laptop:



Observations

- Threads allow us to gain performance.
- More threads doesn't necessarily mean better performance. Why?
- When we create more threads than what we can run in parallel, we create overhead.
- When we use threads we have to pay attention to the overhead.

Sharing Data

- All threads share same data.
- Multiple threads can read data at the same time (e.g., matrix multiplication above).
- Can more than one thread write to the same data at the the same time?
- Can one thread write while another one reads the same data?
- To answer these questions, let's look at an example.

Threads: Example 3

- Let's look at the code snippet below:

```
unsigned long counter = 0;

int numThreads = 10;
vector<thread> threads(numThreads);

for (int j = 0; j < numThreads; ++j)
{
   threads[j] = thread([&counter]()
   {
      for (int i = 0; i<100000; ++i)
      {
         counter++;
      }
   });
}

for (auto& t : threads) t.join();

cout << counter;
```

- To build and run (on Midway):
    1. Get a compute node with 8 CPU cores
       `sinteractive --time=0:30:0 --cpus-per-task=8 --account=finm32950`
    2. Build:
       `icc counter.cpp  -o counter  -lpthread`
    3. Run:
       `./counter`
- What's the value of `counter`?

Counter Example: Possible Scenario 1

- Suppose we have two threads t1 and t2 running in parallel:
  - t1 reads the value of counter, counter is 0
  - t2 reads the value of counter, counter is 0
  - t1 increments it, counter is 1
  - t2 increments it, counter is 1
  - after 2 increments, the value of counter is 1
- When we use threads, we should expect threads to interleave read and modify operations as shown below:

```
      t1                             t2
   ----------                     --------
|   registerA = counter
|                               registerB = counter
|   counter = registerA + 1
|                               counter = registerB + 1
|
|
v (time/clock-cycle)
```

- In this case, the counter will be incremented only once after two add operations.

Counter Example: Possible Scenario 2

- Suppose we have two threads t1 and t2 running in parallel:
  - t1 reads the value of counter, counter is 0
  - t1 increments it, counter is 1
  - t2 reads the value of counter, counter is 1
  - t2 increments it, counter is 2
  - after 2 increments the value of counter is 2

```
t1                           t2
----                        ----
registerA = counter
counter = registerA + 1
                            registerB = counter
                            counter = registerB + 1
```

- In this case, the counter will be incremented twice (correctly) after two add operations.

Race Conditions

- If two (or more) threads write to the same shared data, the result depends on the order in which the threads ran.
- This problem (bug) is known as a *race condition*.
- How do we avoid race conditions?

Critical Region

- The counter is a shared variable in our example.
- Incrementing (modifying) it by two or more threads at the same time leads to a race condition.
- The part where shared resource is accessed is known as the critical section/region

```
for (int j = 0; j < numThreads; ++j)
{
   threads[j] = thread([&counter]()
   {
      for (int i = 0; i<100000; ++i)
      {
         counter++;
      }
   });
}
```

- Only one thread *should* execute the code in the critical section at a time – known as mutual exclusion.

Using a Lock

- One way to achieve mutual exclusion is to use a lock to protect shared data.
- We can use a lock known as mutex (*mut*ual *ex*clusion).
- To protect shared data using a mutex involves:
  - lock the mutex before shared data is accessed
  - unlock the mutex after shared data is accessed
- Only one thread can lock the mutex at a time.
- Everyone else has to wait until the thread holding the mutex releases it.
- `http://en.cppreference.com/w/cpp/thread/mutex`

- We can re-write the counter example using a mutex:

```
for (int j = 0; j < numThreads; ++j)
{
   threads[j] = thread([&]()
   {
       for (int i = 0; i<100000; ++i)
       {
         count_mutex.lock();
         counter++;
         count_mutex.unlock();
      }
   });
}
```

- We are guarding the critical region using a mutex.
- Race condition is avoided.

Deadlocks

- What happens if we lock a mutex and forget to unlock it?
- No thread will be able to access the shared resource again.
- This will lead to a problem, known as a *deadlock*.
- Deadlocks can happen due to exceptions:

```
mutex.lock();

critical_region; //can throw an exception

mutex.unlock();
```

std::lock_guard

- The `std::lock_guard` class implements the RAII technique[4] for a mutex:
  - locks the mutex when the `lock_guard` object is constructed (in the constructor)
  - unlocks the mutex when the `lock_guard` object is destroyed (in the destructor)
- `std::lock_guard`:
  - prevents errors due to forgetting to release a lock
  - guarentees exception safety
- Defined in `<mutex>`

[4] discussed in winter course

- The code snippet below shows how to lock a piece of shared data using a `std::mutex` using this technique.

```
unsigned long count = 0;
std::mutex count_mutex;

int numThreads = 10;
vector<thread> threads(numThreads);

for (int j = 0; j < numThreads; ++j)
{
   threads[j] = thread([&]()
   {
       for (int i = 0; i<100000; ++i)
       {
         std::lock_guard<std::mutex> guard(count_mutex);
         count++;
      }
   });
}
```

Atomics

- Another way to handle critical regions is to use atomic operations.
- Atomic operations are performed as one (all or none) operation so the other threads see the operation as a single operation.
- Atomic operations do not require us to use locks – no need to worry about locking related issues.
- Number of atomic operations are limited – to handle complicated critical regions we still need to use locks.
- Atomic types are defined in atomic header (`http://en.cppreference.com/w/cpp/header/atomic`).

- We can write the counter example using atomics as follows:

```
atomic<unsigned long> counter;
counter = 0;

for (int j = 0; j < numThreads; ++j)
{
   threads[j] = thread([&]()
   {
      for (int i = 0; i<100000; ++i)
      {
         counter++;
      }
   });
}
```