Parallel Graph Algorithms on the Xeon Phi Coprocessor

Master Thesis presentation

Dennis Felsing | 2015-09-07
Motivation: Graphs

- **Complex Network**: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...

Map of the Internet, [http://www.opte.org/maps/](http://www.opte.org/maps/)
Motivation: Graphs

- **Complex Network**: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:
Motivation: Graphs

- **Complex Network**: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:
- **Graph Generation**
  - Create realistic complex networks with generator and parameters
  - Preserves privacy and confidentiality
  - No need to transfer big data
  - Scale to smaller and bigger graphs
Motivation: Graphs

- **Complex Network**: graph with non-trivial topology
- Occur in social networks, cell biology, the internet...
- We consider two existing algorithms:
  - **Graph Generation**
    - Create realistic complex networks with generator and parameters
    - Preserves privacy and confidentiality
    - No need to transfer big data
    - Scale to smaller and bigger graphs
  - **Graph Drawing**
    - Lay out graph visually, mainly for human perception
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging
- Intel Xeon Phi as alternative to the alternative:
  - 60 cores, more than a CPU, fewer than a GPU
  - Similar to program as CPU
  - Viable choice for graph algorithms?
Motivation: Computation

- Data sets grow fast: Internet size doubles every 5 years
- Clock frequency of processors stagnated in last decade
- Instead more parallelism in processors
- Modern GPUs as even more parallel alternative:
  - Massively parallel, thousands of cores
  - Large performance increases with more parallelization
  - General purpose programming more difficult
  - Graph algorithms with irregular data access challenging
- Intel Xeon Phi as alternative to the alternative:
  - 60 cores, more than a CPU, fewer than a GPU
  - Similar to program as CPU
  - Viable choice for graph algorithms?

⇒ Port two graph algorithms to Xeon Phi and evaluate the porting and their performance
Overview

1. Xeon Phi Coprocessor
   - Hardware Architecture
   - Programming

2. Generation of Massive Complex Networks
   - Algorithm
   - Results

3. Graph Drawing using Graph Clustering
   - Algorithm
   - Results
Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)
Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)

<table>
<thead>
<tr>
<th></th>
<th>Host System</th>
<th>Accelerator Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>2 × Xeon E5-2680</td>
<td>Xeon Phi 5110P</td>
</tr>
<tr>
<td>Release Date</td>
<td>2012</td>
<td>2012</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.7 GHz</td>
<td>1.05 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>2 × 8 (32 threads)</td>
<td>60 (240 threads)</td>
</tr>
<tr>
<td>RAM Capacity</td>
<td>256 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>RAM Bandwidth</td>
<td>51 GB/s</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>SIMD Instructions</td>
<td>MMX, SSE, AVX (256 bit)</td>
<td>IMCI (512 bit)</td>
</tr>
</tbody>
</table>
Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)

<table>
<thead>
<tr>
<th></th>
<th>Host System</th>
<th>Accelerator Card</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>2 × Xeon E5-2680</td>
<td>Xeon Phi 5110P</td>
</tr>
<tr>
<td>Release Date</td>
<td>2012</td>
<td>2012</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.7 GHz</td>
<td>1.05 GHz</td>
</tr>
<tr>
<td>Cores</td>
<td>2 × 8 (32 threads)</td>
<td>60 (240 threads)</td>
</tr>
<tr>
<td>RAM Capacity</td>
<td>256 GB</td>
<td>8 GB</td>
</tr>
<tr>
<td>RAM Bandwidth</td>
<td>51 GB/s</td>
<td>320 GB/s</td>
</tr>
<tr>
<td>SIMD Instructions</td>
<td>MMX, SSE, AVX (256 bit)</td>
<td>IMCI (512 bit)</td>
</tr>
</tbody>
</table>

⇒ Parallelization and vectorization necessary to reach high performance
Xeon Phi: Hardware Architecture

- Xeon Phi 5110P used, 60 in-order cores at 1 GHz
- Simple cores based on original Pentium design from 1994
- Augmented with 64-bit support (not x86-64)
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- **Parallelization (Multiple Instruction Multiple Data) methods:**
  - OpenMP
    ```
    float a[MAX], b[MAX], c[MAX];
    
    for (i = 0; i < MAX; i++)
       c[i] = a[i] + b[i];
    ```
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- **Vectorization (Single Instruction Multiple Data) methods**
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- **Parallelization** *(Multiple Instruction Multiple Data)* methods:
  - OpenMP

```c
float a[MAX], b[MAX], c[MAX];
#pragma omp parallel for
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

- OpenMP Offloading
- Cilk Plus
- Threading Building Blocks
- MPI

- **Vectorization** *(Single Instruction Multiple Data)* methods
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods:
  - OpenMP
  - OpenMP Offloading

```c
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic) in(a, b) out(c)
{
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
        c[i] = a[i] + b[i];
}
```

- Cilk Plus
- Threading Building Blocks
- MPI

- Vectorization (Single Instruction Multiple Data) methods
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- **Parallelization (Multiple Instruction Multiple Data) methods:**
  - OpenMP
  - OpenMP Offloading

```c
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic:0) in(a, b) out(c) signal(c)
{
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
        c[i] = a[i] + b[i];
}
#pragma offload_wait target(mic:0) wait(c)
```

- Cilk Plus
- Threading Building Blocks
- MPI

- **Vectorization (Single Instruction Multiple Data) methods**
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: \( c = \_\text{mm512_add_ps}(a, b) \)

512 bits

32 bits (float)

Auto-Vectorization
Cilk Plus
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: $c = \text{mm512_add_ps}(a, b)$

Auto-Vectorization
Cilk Plus
**Xeon Phi: Programming**

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: \[ c = \text{mm512_add_ps}(a, b) \]

512 bits

32 bits (float)

Auto-Vectorization
Cilk Plus
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization: \( c = \text{__mm512_add_ps}(a, b) \)

512 bits

32 bits (float)

Auto-Vectorization
Cilk Plus
**Xeon Phi: Programming**

- Presented as a regular computer, stripped down Linux, SSH
- **Parallelization** *(Multiple Instruction Multiple Data)* methods
- **Vectorization** *(Single Instruction Multiple Data)* methods:
  - Manual Vectorization: `c = _mm512_add_ps(a, b)`

512 bits

32 bits (float)

- 32 bits (float)
- **Auto-Vectorization**
- **Cilk Plus**
Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization (Multiple Instruction Multiple Data) methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization

```c
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));

#pragma simd
for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
```

- Cilk Plus
Network Generation: Algorithm

Desired properties of a complex network:

- **Scale-Free**: No typical vertex degree
  ⇒ Degree distribution follows power law

![Diagram showing degree distribution and power law](image)

- power-law: $x^{-2.8}$
- exponential: $2e^{-0.6x}$
Network Generation: Algorithm

Desired properties of a complex network:

- **Scale-Free**: No typical vertex degree
  - Degree distribution follows power law
- **Small-World**: All nodes connected by short paths

Regular  |  Small-world  |  Random
---|---|---

\[ p = 0 \quad \text{Increasing randomness} \quad p = 1 \]
Network Generation: Algorithm

Desired properties of a complex network:

- **Scale-Free**: No typical vertex degree
  ⇒ Degree distribution follows power law

- **Small-World**: All nodes connected by short paths

⇒ Generator using **hyperbolic geometry** performs well in both properties
Network Generation: Algorithm

- **Exponential expansion of space** in hyperbolic geometry:
  
  Area of circle grows exponentially with distance from center
  
  ⇒ Natural embedding of graphs with tree-like structure
  
  ⇒ May also be good for generating graphs

M.C. Escher: Circle Limit IV
Network Generation: Algorithm

- **Exponential expansion of space** in hyperbolic geometry:
  - Area of circle grows exponentially with distance from center
  - Natural embedding of graphs with tree-like structure
  - May also be good for generating graphs

- **Hyperbolic generator**: Distribute vertices in hyperbolic plane
  - Edge when two vertices are close to each other (hyperbolic circle)
Network Generation: Algorithm

- **Exponential expansion of space** in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs

- **Hyperbolic generator**: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)

- **Poincaré disk model**: Mapping to Euclidean unit disk
  ⇒ Neighborhood transformed to Euclidean circle
Network Generation: Algorithm

- **Exponential expansion of space** in hyperbolic geometry:
  Area of circle grows exponentially with distance from center
  ⇒ Natural embedding of graphs with tree-like structure
  ⇒ May also be good for generating graphs

- **Hyperbolic generator**: Distribute vertices in hyperbolic plane
  Edge when two vertices are close to each other (hyperbolic circle)

- **Poincaré disk model**: Mapping to Euclidean unit disk
  ⇒ Neighborhood transformed to Euclidean circle

- **Polar quadtree**: Efficiently determine neighborhood
Network Generation: Algorithm

- **Exponential expansion of space** in hyperbolic geometry:
  - Area of circle grows exponentially with distance from center
  - Natural embedding of graphs with tree-like structure
  - May also be good for generating graphs

- **Hyperbolic generator**: Distribute vertices in hyperbolic plane
  - Edge when two vertices are close to each other (hyperbolic circle)

- **Poincaré disk model**: Mapping to Euclidean unit disk
  - Neighborhood transformed to Euclidean circle

- **Polar quadtree**: Efficiently determine neighborhood
  - Subquadratic running time $O((n^{3/2} + m) \log n)$
Network Generation: Implementation

- Implementation part of NetworKit, high level C++11 code
- NetworKit ported to Intel C++ compiler 15.0 and Xeon Phi
- Working around compiler restrictions and bugs with constexprs, implicit conversions, null pointers, the standard library, and function traits on lambdas

- Three execution modes implemented and tested:
  - No offloading: Entire code runs on Xeon Phi, not enough memory
  - Full offloading: Offload parts of the calculation, keep results in memory of host system
  - Partial offloading: Offload part of the calculation, other part on the host system
Network Generation: Results

- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (malloc) are locking in default C library glibc
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other
Network Generation: Results

- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (malloc) are locking in default C library glibc
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other

⇒ Use non-locking allocations of Intel’s Threading Building Blocks

Scaling of initial implementation:

![Graph showing speedup factor vs. number of threads for TBB malloc and glibc malloc](image)
Network Generation: Results

- Many memory allocations during algorithm to create dynamically sized lists of neighbors
- Allocations (malloc) are locking in default C library glibc
- On Xeon Phi more threads run in parallel than on CPU, so more allocations block each other

⇒ Use non-locking allocations of Intel’s Threading Building Blocks
⇒ Reduce number of (re)allocations by reusing memory and preallocating expected size

Scaling of initial implementation:

- Scaling of final, optimized code:
Network Generation: Results

- Tuning parameters of the algorithm:
  - **Capacity**: Maximum number of vertices in a leaf cell before split
  - **Balance**: Share of area in outer children when splitting
Network Generation: Results

- Tuning parameters of the algorithm:
  - **Capacity**: Maximum number of vertices in a leaf cell before split
  - **Balance**: Share of area in outer children when splitting

\[ \Rightarrow \text{Imbalanced quadtree with greater space to outer children} \]
Network Generation: Results

- Transferring parts of graph back to host system is slow
  ⇒ Double Buffering on Phi and host

- Buffering for full offloading:

![Graph showing single and double buffer performance](image.png)
Network Generation: Results

- Transferring parts of graph back to host system is slow
  ⇒ Double Buffering on Phi and host

- Buffering for full offloading:

  ![Graph showing running time vs. vertices for single and double buffer]

  ![Graph showing running time vs. vertices for full offload, no offload, and partial offload]

  ⇒ With many optimizations similar speed as dual Xeon, but not faster
Graph Drawing: Algorithm

We assume to have graphs with predefined target edge lengths

- **Full Stress Model**: Physical springs connecting all pairs of vertices
- **Maxent-Stress Model**: Minimize stress, maximize entropy:

\[
M(x) = \sum_{\{u,v\} \in E} w_{uv} (\|x_u - x_v\| - d_{uv})^2 - \alpha \sum_{\{u,v\} \notin E} \ln \|x_u - x_v\|
\]

- **weight factor**
- **target edge length**
- **coordinate vector**
- **stress for target edge lengths**
- **entropy for rest**
Graph Drawing: Algorithm

We assume to have graphs with predefined target edge lengths
- Full Stress Model: Physical springs connecting all pairs of vertices
- Maxent-Stress Model: Minimize stress, maximize entropy:

\[ M(x) = \sum_{\{u,v\} \in E} w_{uv} \left( \|x_u - x_v\| - d_{uv} \right)^2 - \alpha \sum_{\{u,v\} \notin E} \ln \|x_u - x_v\| \]

- Multilevel Maxent-Stress Algorithm:
  - Minimize maxent-stress by clustering graph in multiple levels of hierarchy
  - Contract clusters into new supervertices
  - Iteratively solve maxent-stress on each finer level
  ⇒ Parallelizes well
Graph Drawing: Results

- Preliminary implementation of multilevel maxent-stress graph drawing algorithm
- Parallelized with OpenMP
- Graphs of interest ($< 10^7$ edges) easily fit into Xeon Phi memory ⇒ No expensive offloading necessary
- Source code libraries had to be fixed for ICPC
- No major dynamic allocations in algorithm ⇒ Intel TBB’s `malloc` has small effect
- Inner loop vectorizes well when isolated: speedup factor 7.0 Smaller effect when embedded in real program ⇒ Other calculations at same time, hyper-threading, memory connection busy
## Graph Drawing: Results

<table>
<thead>
<tr>
<th>Graph</th>
<th>$n$</th>
<th>$m$</th>
<th>Description</th>
<th>Phi</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>nyc</td>
<td>264 346</td>
<td>365 050</td>
<td>Road Network</td>
<td>960.0</td>
<td>1845.9</td>
</tr>
<tr>
<td>luxembourg</td>
<td>114 599</td>
<td>119 666</td>
<td>Road Network</td>
<td>89.5</td>
<td>166.5</td>
</tr>
<tr>
<td>commanche</td>
<td>7920</td>
<td>11 880</td>
<td>Helicopter Mesh</td>
<td>2.6</td>
<td>3.5</td>
</tr>
<tr>
<td>rajat06</td>
<td>10 922</td>
<td>18 061</td>
<td>Circuit Simulation</td>
<td>3.5</td>
<td>4.5</td>
</tr>
<tr>
<td>delaunay_n15</td>
<td>32 768</td>
<td>98 274</td>
<td>Delaunay Triangulation</td>
<td>7.8</td>
<td>9.0</td>
</tr>
<tr>
<td>rgg_n_2_15_s0</td>
<td>32 768</td>
<td>160 240</td>
<td>Random Graph</td>
<td>5.3</td>
<td>3.6</td>
</tr>
</tbody>
</table>

![Graph Drawing Graph](image)

- All graphs small enough to fit into Xeon Phi memory
  ⇒ Executed on Xeon Phi directly
- Good speedup for large sparse graphs
Conclusion and Outlook

- Complex algorithms and their framework/libraries ported to Xeon Phi
- Good scaling in both algorithms on Xeon Phi
- Offloading large amounts of data too expensive
- Graph drawing algorithm outperforms two-socket Intel Xeon system, especially on sparse graphs
- Future Research: Direct comparison between graph algorithms on GPU and Xeon Phi
- New Xeon Phi “Knights Landing” this year with modern cores and 384 GB of memory
[Appendix] Xeon Phi: Layout Overview

Diagram showing the layout of Xeon Phi with Core, L2, CRI, TD, PCIe client logic, and GDDR MC.
[Appendix] Xeon Phi: Core Pipeline

32KiB L1 Instruction Cache

Thread 1 → Thread 2 → Thread 3 → Thread 4

Instruction Decode

Execution Pipe 0 → Execution Pipe 1

VPU 512b SIMD

X87

Scalar

32KiB L1 Data Cache

512KiB L2 Cache

Core Ring Interconnect

Tag Directory
Presented as a regular computer, stripped down Linux, SSH

- **Parallelization** methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI

- **Vectorization** (**Single Instruction Multiple Data**) methods
[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
    ```
    float a[MAX], b[MAX], c[MAX];
    
    for (i = 0; i < MAX; i++)
        c[i] = a[i] + b[i];
    ```
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods
[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods:
  - OpenMP
    ```
    float a[MAX], b[MAX], c[MAX];
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
      c[i] = a[i] + b[i];
    ```
  - OpenMP Offloading
  - Cilk Plus
  - Threading Building Blocks
  - MPI
- Vectorization (Single Instruction Multiple Data) methods
Presented as a regular computer, stripped down Linux, SSH

- **Parallelization methods:**
  - OpenMP
  - OpenMP Offloading

```c
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic) in(a, b) out(c)
{
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
        c[i] = a[i] + b[i];
}
```

- Cilk Plus
- Threading Building Blocks
- MPI

- **Vectorization** *(Single Instruction Multiple Data)* methods
Presented as a regular computer, stripped down Linux, SSH

- **Parallelization methods:**
  - OpenMP
  - OpenMP Offloading

```c
float a[MAX], b[MAX], c[MAX];
#pragma offload target(mic:0) in(a, b) out(c) signal(c)
{
    #pragma omp parallel for
    for (i = 0; i < MAX; i++)
        c[i] = a[i] + b[i];
}
#pragma offload_wait target(mic:0) wait(c)
```

- Cilk Plus
- Threading Building Blocks
- MPI

- **Vectorization (Single Instruction Multiple Data) methods**
[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- **Parallelization** methods:
  - OpenMP
  - OpenMP Offloading
  - Cilk Plus

  ```c
  float a[MAX], b[MAX], c[MAX];
  cilk_for (i = 0; i < MAX; i++)
  c[i] = a[i] + b[i];
  ```

- Threading Building Blocks
- MPI

- **Vectorization** (**Single Instruction Multiple Data**) methods
Presented as a regular computer, stripped down Linux, SSH

Parallelization methods:
- OpenMP
- OpenMP Offloading
- Cilk Plus
- Threading Building Blocks
  ```c
  float a[MAX], b[MAX], c[MAX];
  parallel_for(size_t(0), MAX, size_t(1), [=](size_t i) {
    c[i] = a[i] + b[i];
  });
  ```
- MPI

Vectorization (Single Instruction Multiple Data) methods
Presented as a regular computer, stripped down Linux, SSH

Parallelization methods

Vectorization (Single Instruction Multiple Data) methods:

- Manual Vectorization: \[ c = \text{__mm512_add_ps}(a, b) \]
  ```c
  #include <immintrin.h>
  float a[MAX] __attribute__((aligned(64)));
  float b[MAX] __attribute__((aligned(64)));
  float c[MAX] __attribute__((aligned(64)));
  __m512 _a, _b, _c;
  for (i = 0; i < MAX; i += 16) {
    _a = __mm512_load_ps(&a[i]);
    _b = __mm512_load_ps(&b[i]);
    _c = __mm512_add_ps(_a, _b);
    __mm512_store_ps(&c[i], _c);
  }
  ```

- Auto-Vectorization

- Cilk Plus
[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization

```c
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
#pragma simd
for (i = 0; i < MAX; i++)
    c[i] = a[i] + b[i];
```
- Cilk Plus
[Appendix] Xeon Phi: Programming

- Presented as a regular computer, stripped down Linux, SSH
- Parallelization methods
- Vectorization (Single Instruction Multiple Data) methods:
  - Manual Vectorization
  - Auto-Vectorization
  - Cilk Plus

```c
float a[MAX] __attribute__((aligned(64)));
float b[MAX] __attribute__((aligned(64)));
float c[MAX] __attribute__((aligned(64)));
c[i:MAX] = a[i:MAX] + b[i:MAX];
```
**Appendix** Preliminaries: Graphs

- Graph $G = (V, E)$ consists of
  - Set of vertices $V$
  - Set of edges $E \subseteq V \times V$
- Edge $e = (u, v) \in E$: connection from source $u$ to target $v$
- Number of vertices $n = |V|$
- Number of edges $m = |E|$
- Undirected graphs only: $(u, v) \in E$ iff $(v, u) \in E$
- Neighborhood $N(u) = \{v : (u, v) \in E\}$
- Loop $(u, u) \in E$
- Degree $deg(v)$: number of incident edges, counting loops twice
- Distance: number of edges in shortest path connecting vertices
- Diameter $d$: greatest distance between any pair of vertices
OpenMP Scheduling:

- **Compact** thread affinity
  
- **Scatter** thread affinity

⇒ Scatter and balanced 1.4 times faster