DAY 6: OPTIMIZATION ON PARALLEL INTEL® ARCHITECTURES

Lecture day 6

Ryo Asai
Colfax International — colfaxresearch.com

April 2017
While best efforts have been used in preparing this training, Colfax International makes no representations or warranties of any kind and assumes no liabilities of any kind with respect to the accuracy or completeness of the contents and specifically disclaims any implied warranties of merchantability or fitness of use for a particular purpose. The publisher shall not be held liable or responsible to any person or entity with respect to any loss or incidental or consequential damages caused, or alleged to have been caused, directly or indirectly, by the information or programs contained herein. No warranty may be created or extended by sales representatives or written sales materials.
Introduction to Intel DAAL, Part 2: Polynomial Regression with Batch Mode Computation

Parallel Programming Book
Introduction to parallel programming, deep discussions of optimization techniques, etc.
Optimization Techniques for the Intel MIC Architecture, Part 2 of 2: Software Sharding and Pudding

Software Developer’s Introduction to the IN3T Ultrastar Archive HNS SNR Drives
Optimization Techniques for the Intel MIC Architecture, Part 2 of 2: Software Sharding and Pudding
Optimization Techniques for the Intel MIC Architecture, Part 2 of 2: Software Sharding and Pudding

Performance in Power and Performance in the Cloud with Intel Xeon Phi Coprocessors (and why it is so difficult to measure)

Welcome
http://colfaxresearch.com/
Code Modernization

Optimizing software to better utilize features available in modern computer architectures.

- Scalar Tuning
- Vectorization
- Threading
- Memory
- Communication

Node-level
Cluster-level
§1. QUICK REVIEW
**PARALLELISM**

Task Parallelism – multiple instructions multiple data elements (MIMD)

Data Parallelism – single instruction multiple data elements (SIMD)

Unbounded growth opportunity, but **not automatic**
Vector instructions – one of the implementations of SIMD (Single Instruction Multiple Data) parallelism.

<table>
<thead>
<tr>
<th>Scalar Instructions</th>
<th>Vector Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 + 1 = 5</td>
<td>4 + 1 = 5</td>
</tr>
<tr>
<td>0 + 3 = 3</td>
<td>0 + 3 = 3</td>
</tr>
<tr>
<td>-2 + 8 = 6</td>
<td>-2 + 8 = 6</td>
</tr>
<tr>
<td>9 + -7 = 2</td>
<td>9 + -7 = 2</td>
</tr>
</tbody>
</table>

Vector Length
I have a vectorized and multi-threaded code!

Some people stop here. But even if your application is multi-threaded and vectorized, it may not be optimal. Optimization could unlock more performance for your application.

Example areas for consideration:

▷ Multi-threading
  • Do my threads have enough work?
  • Are my threads independent?
  • Is work distributed properly?

▷ Vectorization
  • Is my data organized well for vectorization?
  • Do I have regular loop patterns?
§2. OPTIMIZATION EXAMPLES
EXAMPLE: NUMERICAL INTEGRATION

\[ I(a, b) = \int_0^a \frac{1}{\sqrt{x}} \, dx \]

Rectangle method:

\[ \Delta x = \frac{a}{n}, \]

\[ x_i = (i + 1)\Delta x, \]

\[ I(a, b) = \sum_{i=0}^{n-1} \frac{1}{\sqrt{x_i}} \Delta x + O(\Delta x). \]

```cpp
float Integrate(const float a,
                const int N) {
    const float dx = a/float(n);
    float S = 0.0f;
    for (int i = 0; i < n; i++) {
        const float xi = dx*float(i+1);
        S += 1.0f/sqrtf(xi) * dx;
    }
    return S;
}
```
Incorrect: Race condition on integral

```cpp
const double dx = (x_upper_bound - x_lower_bound)/nSteps;
double integral = 0.0;

#pragma omp parallel for
for(int i = 0; i < nSteps; i++) {
    const double x = x_lower_bound + dx*(double(i) + 0.5);
    integral += 1.0/sqrt(x) * dx;
}
```

Non-optimal: Too much synchronization. Slower than serial implementation.

```cpp
#pragma omp parallel for
for(int i = 0; i < nSteps; i++) {
    const double x = x_lower_bound + dx*(double(i) + 0.5);
    #pragma omp atomic
    integral += 1.0/sqrt(x) * dx;
}
```
Manual reduction implementation

```c
    double integral = 0.0;
    #pragma omp parallel
    {
        double integral_th=0.0;
        #pragma omp for
        for(int i = 0; i < nSteps; i++) {
            const double x = x_lower_bound + dx*(double(i) + 0.5);
            integral_th += 1.0/sqrt(x) * dx;
        }
        #pragma omp atomic
        integral += integral_th;
    }
```

OpenMP reduction. Array support from 4.5.

```c
    #pragma omp parallel for reduction(+:integral)
```
\[ \Phi (\vec{R}_j) = - \sum_{i=1}^{m} \frac{q_i}{|\vec{r}_i - \vec{R}_j|} , \]  

\[ |\vec{r}_i - \vec{R}| = \sqrt{(r_{i,x} - R_x)^2 + (r_{i,y} - R_y)^2 + (r_{i,z} - R_z)^2} . \]
```c
struct Charge {
    float x, y, z, q;
};

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++) {
            const float Rx = (float)i, Ry = (float)j, Rz = (float)k;
            float phi_p = 0.0;
            for (int l=0; l<m; l++) {
                const float dx=chg[l].x - Rx;
                const float dy=chg[l].y - Ry;
                const float dz=chg[l].z - Rz;
                phi_p -= chg[l].q / (dx*dx+dy*dy+dz*dz); // Coulomb’s law
            }
            phi[i*n*n+j*n+k] = phi_p;
        }
```
Bad: parallelizing the inner loop increases overhead of OpenMP

```c
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        #pragma omp parallel for
        for (int k = 0; k < n; k++) {
```

Better: if \( n \) is small, not enough work to occupy all threads.

```c
#pragma omp parallel for
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++) {
```

Good: collapse loops means threads have more iterations to work with.

```c
#pragma omp parallel for collapse(2)
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++) {
```
Idea: combine iterations spaces of the inner loop and the outer loop.

```c
#pragma omp parallel for collapse(2)
for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++) {
        // ...
        // ...
    }
```

```c
#pragma omp parallel for
for (int c = 0; c < m*n; c++) {
    i = c / n;
    j = c % n;
    // ...
}
```
## Loop Scheduling Modes in OpenMP

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>0</td>
<td>0 1 2 3 4 5 6 7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>8 9 10 11 12 13 14 15</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16 17 18 19 20 21 22 23</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24 25 26 27 28 29 30 31</td>
</tr>
</tbody>
</table>

| dynamic    | 0       | 0 7 10 12 17 ... |
|            | 1       | 1 4 9 14 18 ... |
|            | 2       | 2 5 8 11 15 ... |
|            | 3       | 3 6 9 13 16 ... |

| guided      | 0       | 0 1 2 3 16 17 24 29 |
|             | 1       | 4 5 6 7 20 21 25 30 |
|             | 2       | 8 9 10 11 18 19 26 28 |
|             | 3       | 12 13 14 15 22 23 27 31 |

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Threads</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>static,1</td>
<td>0</td>
<td>0 4 8 12 16 20 24 28</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1 5 9 13 17 21 25 29</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2 6 10 14 18 22 26 30</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3 7 11 15 19 23 27 31</td>
</tr>
</tbody>
</table>

| dynamic,2  | 0       | 0 1 10 11 16 17 ... |
|            | 1       | 2 3 12 13 18 19 ... |
|            | 2       | 4 5 8 9 14 15 ... |
|            | 3       | 6 7 20 21 ... |

| guided,2   | 0       | 0 1 2 3 16 17 24 25 |
|            | 1       | 4 5 6 7 20 21 25 27 |
|            | 2       | 8 9 10 11 18 19 28 29 |
|            | 3       | 12 13 14 15 22 23 30 31 |
To set scheduling for a particular loop in code (example):

```
#pragma omp parallel for schedule(dynamic,4)
   // ...
```

To set scheduling for the entire application at run time (example):

```
#pragma omp parallel for schedule(runtime)
   // ...
```

```
vega@lyra% export OMP_SCHEDULE=dynamic,4
vega@lyra% ./run-my-app
```
Arrays of Structures versus Structures of Arrays

Array of Structures (AoS)

```c
struct Charge {  // Elegant, but ineffective data layout
    float x, y, z, q;  // Coordinates and value of this charge
};
// The following line declares a set of m point charges:
Charge chg[m];
```

Structure of Arrays (SoA)

```c
struct Charge_Distribution {
    // Data layout permits effective vectorization of Coulomb’s law application
    const int m;  // Number of charges
    float * x;  // Array of x-coordinates of charges
    float * y;  // ...y-coordinates...
    float * z;  // ...etc.
    float * q;  // These arrays are allocated in the constructor
};
```
Arrays of Structures versus Structures of Arrays

Array of Structures (sub-optimal)

Structure of Arrays (optimal)

Optimization Examples

© Colfax International, 2013–2017
**Optimized Solution: Structure of Arrays, Unit-Stride Access**

```c
struct Charge_Distribution {
    // Data layout permits effective vectorization of Coulomb’s law application
    const int m; // Number of charges
    float *x, *y, *z, *q; // Arrays of x-, y- and z-coordinates of charges
};
```

// This version vectorizes better thanks to unit-stride data access
for (int i=0; i<chg.m; i++) {
    // Unit stride: (&chg.x[i+1] - &chg.x[i]) == sizeof(float)
    const float dx=chg.x[i] - Rx;
    const float dy=chg.y[i] - Ry;
    const float dz=chg.z[i] - Rz;
    phi -= chg.q[i] / sqrtf(dx*dx+dy*dy+dz*dz);
}
§3. ADDITIONAL TOPIC: UNROLL AND JAM
for (int i = 0; i < m; i++) // Original code:
    for (int j = 0; j < n; j++)
        compute(a[i], b[j]); // Memory access is unit-stride in j

// Step 1: strip-mine outer loop
for (int ii = 0; ii < m; ii += TILE)
    for (int i = ii; i < ii + TILE; i++)
        for (int j = 0; j < n; j++)
            compute(a[i], b[j]); // Same order of operation as original

// Step 2: permute and vectorize outer loop
for (int ii = 0; ii < m; ii += TILE)
    #pragma simd
    for (int j = 0; j < n; j++)
        for (int i = ii; i < ii + TILE; i++)
            compute(a[i], b[j]); // Use each vector in b[j] a total of TILE times

© Colfax International, 2013–2017
for (int k = 0; k < n; k+=4) {
    const float Rx = (float) i, Ry = (float) j, Rz1 = (float) k,
                    Rz2 = (float) k+1, Rz3 = (float) k+2, Rz4 = (float) k+3;
    float phi_p1 = 0.0, phi_p2 = 0.0, phi_p3 = 0.0, phi_p4 = 0.0;
    #pragma omp simd reduction(+: phi_p1, phi_p2, phi_p3, phi_p4)
    for (int l=0; l<m; l++) {
        const float dx =chg.x[l] - Rx,   dy=chg.y[l] - Ry;
        dz1=chg.z[l] - Rz1,   dz2=chg.z[l] - Rz2,
        dz3=chg.z[l] - Rz3,   dz4=chg.z[l] - Rz4;
        phi_p1 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz1*dz1);
        phi_p2 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz2*dz2);
        phi_p3 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz3*dz3);
        phi_p4 -= chg.q[l] / sqrtf(dx*dx+dy*dy+dz4*dz4);
    }
    phi[i*n*n+j*n+k]   = phi_p1; phi[i*n*n+j*n+k+1] = phi_p2;
    phi[i*n*n+j*n+k+2] = phi_p3; phi[i*n*n+j*n+k+3] = phi_p4;
} // Remainders to follow (dealing with case k%4 != 0) ...
§4. MEMORY CONSIDERATIONS
FLOPS VS BANDWIDTH
Theoretical estimates, Intel Xeon E5-2697 V3 processor

Performance = 28 cores × 2.7 GHz × (256/64) vec.lanes × 2 FMA × 2 FPU ≈ 1.2 TFLOP/s

Required Data Rate = 1.2 TFLOP/s × 8 bytes ≈ 10 TB/s

Memory Bandwidth = η × 2 × 59.7 ≈ 0.1 TB/s

Ratio = 10/0.1 ≈ 100 (FLOPs)/(Memory Access)

If the Arithmetic Intensity is...

➢ > 100 (FLOPs)/(Memory Access) — Compute Bound Application

➢ < 100 (FLOPs)/(Memory Access) — Bandwidth Bound Application
More on roofline model: Williams et al.
### ON COMPUTATIONAL COMPLEXITY OF ALGORITHMS

<table>
<thead>
<tr>
<th>Type</th>
<th>Properties</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(N)$</td>
<td>Each data element is used a fixed number of times. Memory-bound unless the number of times is large.</td>
<td>Array scaling, image brightness adjustment, vector dot-product.</td>
</tr>
<tr>
<td>$O(N^\alpha)$</td>
<td>Each element is used $N^{\alpha-1}$ times. A lot of data reuse for $\alpha &gt; 1$. Good implementation can be compute-bound, poor one – memory-bound.</td>
<td>Matrix-matrix multiplication: $O(N^{3/2})$ ($N =$ amount of data in matrix), direct N-body calculation: $O(N^2)$</td>
</tr>
<tr>
<td>$O(N \log N)$</td>
<td>Each element is used $\log N$ times. For small problems – memory-bound, for very large problems transitions to compute-bound</td>
<td>Fast Fourier transform, merge sort</td>
</tr>
<tr>
<td>$O(\log N)$</td>
<td>Always memory-bound.</td>
<td>Binary search</td>
</tr>
</tbody>
</table>

$N =$ data size
NUMA ARCHITECTURE
NUMA = Non-Uniform Memory Access. Cores have fast access to local memory, slow access to remote memory.

Examples:
- Multi-socket Intel Xeon processors
- Second generation Intel Xeon Phi in sub-NUMA clustering mode
Hierarchical cache structure

Two-way processors have NUMA architecture

Intel Xeon CPU: Memory Organization

- **Hierarchical cache structure**
- **Two-way processors have NUMA architecture**
WORKING WITH NUMA ARCHITECTURES
BINDING TO NUMA NODES WITH `numactl`

- `libnuma` – a Linux library for fine-grained control over NUMA policy
- `numactl` – a tool for global NUMA policy control

```bash
vega@lyra% numactl --hardware
available: 2 nodes (0-1)
node 0 cpus: 0 1 2 3 4 5 12 13 14 15 16 17
node 0 size: 65457 MB
node 0 free: 24426 MB
node 1 cpus: 6 7 8 9 10 11 18 19 20 21 22 23
node 1 size: 65536 MB
node 1 free: 53725 MB
node distances:
node  0  1
  0: 10 21
  1: 21 10
vega@lyra% numactl --membind=<nodes> --cpunodebind=<nodes> ./myApplication
```
WHAT IS THREAD AFFINITY

- OpenMP threads may migrate between cores
- Forbid migration — improve locality — increase performance
- Affinity patterns “scatter” and “compact” may improve cache sharing, relieve thread contention
OMP_PROC_BIND AND OMP_PLACES VARIABLES

Control the binding pattern, including nested parallelism:

```
OMP_PROC_BIND=type[,type[,...]]
```

Here type=true, false, spread, close or master. Comma separates settings for different levels of nesting (OMP_NESTED must be enabled).

Control the granularity of binding:

```
OMP_PLACES=<threads|cores|sockets|(explicit)>
```
Allocation on First Touch

- Memory allocation occurs not during \_mm\_malloc\(), but upon the first write to the buffer (“first touch”)
- Default NUMA allocation policy is “on first touch”
- For better performance in NUMA systems, initialize data with the same parallel pattern as during data usage

```c
float* A = (float*)\_mm\_malloc(n*m*sizeof(float), 64);

// Initializing from parallel region for better performance
#pragma omp parallel for
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        A[i*m + j] = 0.0f;
```
**FIRST-TOUCH ALLOCATION POLICY**

### Poor First-Touch Allocation

- **Array**: $A[i]
- **Memory**: VM page 0, VM page 1, VM page 2, VM page 3
- **CPU 0**: Serial execution
  - For loop: $i = 0$ to $n$
  - $A[i] = 0.0$

### Good First-Touch Allocation

- **Array**: $A[i]
- **Memory**: VM page 0, VM page 1, VM page 2, VM page 3
- **CPU 0**: Parallel execution
  - Thread 0: $i = 0$ to $n/4$
  - $A[i] = 0.0$
- **CPU 1**: Parallel execution
  - Thread 1: $i = n/4$ to $n/2$
  - $A[i] = 0.0$
  - Thread 2: $i = n/2$ to $3n/4$
  - $A[i] = 0.0$
  - Thread 3: $i = 3n/4$ to $n$
  - $A[i] = 0.0$

---

**colfaxresearch.com/**

© Colfax International, 2013–2017
Impact of First-Touch Allocation

Vectorized Parallel Code (Private Variables)

- Intel Xeon processor E5-2697 V2
- Intel Xeon Phi coprocessor 7120P (KNC)
- Intel Xeon Phi processor 7210 (KNL)

Parallel Initialization (First-Touch Allocation)

- Intel Xeon processor E5-2697 V2
- Intel Xeon Phi coprocessor 7120P (KNC)
- Intel Xeon Phi processor 7210 (KNL)

Performance, billion values/s (higher is better)