Lecture 4

Performance: memory system, programming, measurement and metrics

Vectorization
Announcements

• Dirac accounts
  www-ucsd.edu/classes/fa12/cse260-b/Dirac.html

• Mac Mini Lab (with GPU)
  • Available for remote login 6pm to 3am every day
  • Lab sessions in APM: Tuesdays and Fridays 4-6p
Today’s lecture

• Performance of stencil methods

• Vectorization

• Memory system
  ❖ Nehalem
  ❖ Phenom

• Performance
  ❖ Measurement
  ❖ Metrics
Motivating application

• Solve Laplace’s equation in 3-D with Dirichlet Boundary conditions
  \[ \Delta u = 0, \quad u = f \text{ on } \partial \Omega \]

• Building block: iterative solver using Jacobi’s method (7-point stencil)

\[
\text{for } (i,j,k) \text{ in } 1:N \times 1:N \times 1:N \\
u'[i,j,k] = (u[i-1,j,k] + u[i+1,j,k] + \\
u[i,j-1,k] + u[i,j+1,k] + \\
u[i,j,k+1] + u[i,j,k-1]) / 6.0
\]
Memory representation and access

3D grid

Linear array space
Memory access patterns – 3D
Interactions between threads
Interactions between threads

- Thread sharing
- False sharing
Processor Geometry

![Graph showing speedup for different processor geometries and node counts.](image)
Memory access patterns – 2D
False sharing and conflict misses

- Boundary values, false sharing, internal fragmentation
- Large memory access strides, conflict misses
- Compare with distributed memory solution

On a single processor

On multiple processors

Parallel Computer Architecture, Culler, Singh, & Gupta
Reducing conflict misses

- Pad the array with unused cells to change the memory access patterns
- Rivera & Tseng [ICS ‘99, SC ‘00]
Today’s lecture

• Performance of stencil methods
• Vectorization
• Memory system
  - Nehalem
  - Phenom
• Performance
  - Measurement
  - Metrics
Vector processing
Streaming SIMD Extensions

- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (AVX: 256 bits)

for $i = 0:N-1$ \{ $p[i] = a[i] * b[i]$; \}

4 floats
2 doubles
16 bytes
Fused Multiply/Add

\[ r[0:3] = c[0:3] + a[0:3] \times b[0:3] \]

Courtesy of Mercury Computer Systems, Inc.
How do we use the SSE instructions?

• Low level: assembly language or libraries
• Higher level: a vectorizing compiler
  
  gcc -O3 -ftree-vectorizer-verbose=2  s35.c

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

  s35.c:8: note: LOOP VECTORIZED.
  s35.c:4: note: vectorized 1 loops in function
How does non-vectorized code compare?

• Low level: assembly language or libraries
• Higher level: a vectorizing compiler

```c
#pragma novector
for (int i=0; i<N; i++)    //  N = 2048 * 1024
    a[i] = b[i] + c[i];
```

Single precision, running on a Nehalem processor using pgcc

- With vectorization: 7.693 sec.
- Without vectorization: 10.17 sec.

• Double precision

- With vectorization: 11.88 sec.
- Without vectorization: 11.85 sec.
How does the vectorizer work?

- Transformed code
  
  ```c
  for (i = 0; i < 1024; i+=4)
    a[i:i+3] = b[i:i+3] + c[i:i+3];
  ```

- Vector instructions
  
  ```c
  for (i = 0; i < 1024; i+=4){
    vB = vec_ld( &b[i] );
    vC = vec_ld( &c[i] );
    vA = vec_add( vB, vC );
    vec_st( vA, &a[i] );
  }
  ```
What prevents vectorization

• Data dependencies
  for (int i = 1; i < N; i++)
    b[i] = b[i-1] + 2;

  Loop not vectorized: data dependency

• Inner loops only
  for (int j=0; j< reps; j++)
    for (int i=0; i<N; i++)
      a[i] = b[i] + c[i];
What prevents vectorization

- Interrupted flow out of the loop
  
  ```
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
  }
  ```

  Loop not vectorized/parallelized: multiple exits

- This loop will vectorize
  
  ```
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
  }
  ```
Run-time data dependence testing

- Restrict keyword needed to ensure correct semantics
  
  "During the scope of the pointer declaration, all data accessed through it will be accessed .. through any other pointer... [thus] a given object cannot be changed through another pointer."

```
icpc -O2 -vec-report3 -restrict t2a.c

void copy_conserve(char *restrict p, char *restrict q, int n) {
    int i;
    if (p+n < q || q+n < p)
        #pragma ivdep
        for (i = 0; i < n; i++) p[i] = q[i]; /* vector loop */
    else
        for (i = 0; i < n; i++) p[i] = q[i]; /* serial loop */
}

copy.c(11): (col. 3) remark: LOOP WAS VECTORIZED.
```
Alignment

- Unaligned data movement is expensive
- Accesses aligned on 16 byte boundaries go faster
- Intel compiler can handle some alignments
  
  [Link to Dr. Dobbs](http://drdobbs.com/cpp/184401611)

```c
double a[N], b[N];
for (int i = 1; i < N-1; i++)
a[i+1] = b[i] * 3;
```

**cannot be vectorized:**

```c
void fill (char *x)
{
  for (int i = 0; i < 1024; i++)
    x[i] = 1;
}
```

```c
for (int i = 2; i < N-1; i++)
a[i+1] = b[i] * 3
```

```c
peel = x & 0x0f;
if (peel != 0) {
  peel = 16 - peel;
  for (i = 0; i < peel; i++) x[i] = 1;
}
/* aligned access */
for (i = peel; i < 1024; i++) x[i] = 1;
```
Today’s lecture

• Performance of stencil methods
• Vectorization
• Memory system
  - Nehalem
  - Phenom
• Performance
  - Measurement
  - Metrics
Nehalem’s memory hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual*

### Table 2-7. Cache Parameters of Intel Core i7 Processors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First Level Data</td>
<td>32 KB</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>1</td>
<td>Writeback</td>
</tr>
<tr>
<td>Instruction</td>
<td>32 KB</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Second Level</td>
<td>256 KB</td>
<td>8</td>
<td>64</td>
<td>10^1</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
<tr>
<td>Third Level (Shared L3)^2</td>
<td>8MB</td>
<td>16</td>
<td>64</td>
<td>35-40+^2</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.
AMD Phenom’s memory hierarchy

**Dedicated L1**
- Locality keeps most critical data in the L1 cache
- Lowest latency
- 2 loads per cycle

**Dedicated L2**
- Sized to accommodate the majority of working sets today
- Dedicated to eliminate conflicts common in shared caches

**Shared L3 – NEW**
- Victim-cache architecture maximizes efficiency of cache hierarchy
- Fills from L3 leave likely shared lines in the L3
- Sharing-aware replacement policy

---

6MB L3 on cseclass 01 and 02


10/16/12 ©2012 Scott B. Baden /CSE 260/ Fall 2012 27
NUMA awareness

• When we allocate memory, a NUMA processor uses the *first-touch policy*, unless told otherwise.

• Be sure to use `parallel for` when initializing data:

```c
x = new double[n], y,z = new double[n];
#pragma omp parallel for
for (i=0; i<n; i++) {x[i]=0; y[i] = …; z[i] = …}
#pragma omp parallel for
for (i=0; i<n; i++) {x[i]  = y[i] + z[i];
```

©2012 Scott B. Baden /CSE 260/ Fall 2012
Today’s lecture

• Performance of stencil methods
• Vectorization
• Memory system
  ◆ Nehalem
  ◆ Phenom
• Performance
  ◆ Measurement
  ◆ Metrics
Measuring performance
Challenges to measuring performance

- Reproducibility
  - Transient system operating conditions
  - Differing systems or program configuration
- Measurements are imprecise
  - “Heisenberg uncertainty principle:” measurement technique may affect performance
  - Overheads and inaccuracy
- Explain anomalous behavior, but ignore anomalies that are not significant
Complications

- Cost of measuring a full run is prohibitive
  - Ignore startup code if you plan to run for a much longer time in production
- Transient behavior
  - Repeat your measurements
  - “Warm up” the code before collecting measurements
  - Ignore outliers unless their behavior is important to you
  - Average time, maximum time, minimum time?
Measurement collection

- Report the *best* timings
  - Repeat results $\times 3$ to 5 until at least 2 measures agree to within… 5%, 10%
  - Report the minimum time
- Also report outliers
- A scatter plot or error bar can be useful
Why do we take the minimum time?
Measurement errors are not distributed symmetrically.
Timing collection

- **Measures of time**
  - Elapsed, or “wall clock” time
  - CPU time = system + user time
  - Overhead, resolution, and quantization effects

- **Measurement tools**
  - Can be platform dependent, especially library routines
  - Unix `time` command does a reasonable job for long-running programs
  - `gettimeofday()`
Enable others to reproduce your results

• Builds confidence within a community
• Report where you ran, software versions, processor, etc.
  ► `uname -a`
    ► Linux lilliput 2.6.35-30-server #61-Ubuntu SMP Tue Oct 11 18:09:44 UTC 2011 x86_64 GNU/Linux
  ► `gcc --version`
    gcc version 4.4.5 (Ubuntu/Linaro 4.4.4-14ubuntu5)
  ► `icpc --version`
    icpc (ICC) 12.0.2 20110112
  ► `nvcc --version`
    Cuda compilation tools, release 4.0, V0.2.1221
  ► Access processor configuration information
    ► Device # 0 has 30 cores
    ► Device # 1 has 4 cores
    ► Choosing device 0
    ► Device is a GeForce GTX 285, capability: 1.3
    ► CUDA Driver version: 2030, runtime version: 2030

10/16/12 ©2012 Scott B. Baden /CSE 260/ Fall 2012 37
Performance metrics
Measures of Performance

• Why do we measure performance?
• Measures of performance
   Completion time
   Processor time product
    Completion time × # processors
   Throughput: amount of work that can be accomplished in a given amount of time
   Relative performance: given a reference architecture or implementation
    AKA Speedup
Parallel Speedup and Efficiency

• How much of an improvement did our parallel algorithm obtain over the serial algorithm?
• Define the *parallel speedup*, $S_P$

$$S_P = \frac{\text{Running time of the best serial program on 1 processor}}{\text{Running time of the parallel program on P processors}}$$

• $T_1$ is defined as the running time of the “best serial algorithm”
• In general: *not* the running time of the parallel algorithm on 1 processor
• **Definition**: *Parallel efficiency* $E_P = S_P/P$
What can go wrong with speedup?

• Not always an accurate way to compare different algorithms….
• .. or the same algorithm running on different machines
• We might be able to obtain a better running time even if we lower the speedup
• For an individual user the bottom line is running time $T_p$ or the \textit{space time cost} $P T_p$
Superlinear speedup

• We have a super-linear speedup when

\[ S_P > P \Rightarrow E_P > 1 \]

• Super-linear speedups are often an artifact of inappropriate measurement technique

• Where there is a super-linear speedup, a better serial algorithm may be lurking
Scalability

- A computation is **scalable** if performance increases as a “nice function” of the number of processors, e.g. linearly.
- In practice scalability can be hard to achieve:
  - Serial sections: code that runs on only one processor
  - “Non-productive” work associated with parallel execution, e.g. communication
  - Load imbalance: uneven work assignments over the processors
- Some algorithms present intrinsic barriers to scalability leading to alternatives:
  \[
  \text{for } i=0:n-1 \text{ sum = sum + x[i]}
  \]
Serial Section

• Limits scalability
• Let $f = $ the fraction of $T_1$ that runs serially
• $T_1 = f \times T_1 + (1-f) \times T_1$
• $T_p = f \times T_1 + (1-f) \times T_1 / P$

Thus $S_p = 1/[f + (1 - f)/p]$

• As $P \rightarrow \infty$, $S_p \rightarrow 1/f$
• This is known as *Amdahl’s Law* (1967)
Amdahl’s law (1967)

- A serial section limits scalability
- Let \( f = \) fraction of \( T_1 \) that runs serially
- *Amdahl's Law (1967)*: As \( P \to \infty \), \( S_P \to 1/f \)
Weak scaling

• Is Amdahl’s law pessimistic?
• Observation: Amdahl’s law assumes that the workload ($W$) remains fixed
• But parallel computers are used to tackle more ambitious workloads
• If we increase $W$ with $P$ we have weak scaling
  $f$ often decreases with $W$
Computing scaled speedup

• Instead of asking what the speedup is, let’s ask how long a parallel program would run on a single processor [J. Gustafson 1992]

• Let $T_P = 1$
• $f' = \text{fraction of serial time spent on the parallel program}$
• $T_1 = f' + (1 - f') \times P = S'_P = \text{scaled speedup}$
• Scaled speedup is linear in $P$
Isoefficiency

- Consequence of Gustafson’s observation is that we increase N with P
- Kumar: We can maintain constant efficiency so long as we increase N appropriately
- The *isoefficiency* function specifies the growth of N in terms of P
- If N is linear in P, we have a scalable computation
- Problem: the amount of memory per core is shrinking
Fin