CSCI-580 Advanced High Performance Computing

Memory Optimization

Bo Wu
Colorado School of Mines

Part of slides from Keith Cooper & Linda Cooper (Rice University) and Michelle Strout (Colorado State University)
Copyright 2011, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 512 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
Widening Gap between CPU and GPU

Moore’s law effect

- CPU performance
  - Gap grows at 50% per year
- Memory performance
  - 7%/yr
- 60%/yr

Performance

Time
Energy Concern

50X perf
2X power!

70% power will be on data movement
Predicted Performance Numbers from DOE

<table>
<thead>
<tr>
<th>Systems</th>
<th>2009</th>
<th>2011</th>
<th>2015</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Peak Flops/s</td>
<td>2 Peta</td>
<td>20 Peta</td>
<td>1 Exa</td>
<td></td>
</tr>
<tr>
<td>System Memory</td>
<td>0.3 PB</td>
<td>1 PB</td>
<td>5 PB</td>
<td>10 PB</td>
</tr>
<tr>
<td>Node Performance</td>
<td>125 GF</td>
<td>200 GF</td>
<td>400 GF</td>
<td>1-10 TF</td>
</tr>
<tr>
<td>Node Memory BW</td>
<td>25 GB/s</td>
<td>40 GB/s</td>
<td>100 GB/s</td>
<td>200-400 GB/s</td>
</tr>
<tr>
<td>Node Concurrency</td>
<td>12</td>
<td>32</td>
<td>0(100)</td>
<td>0(1000)</td>
</tr>
</tbody>
</table>

Memory Optimization is critical for perf & power.

<table>
<thead>
<tr>
<th>Total Concurrency</th>
<th>225,000</th>
<th>3 Million</th>
<th>0(Billion)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>15 PB</td>
<td>30 PB</td>
<td>150 PB</td>
</tr>
<tr>
<td>I/O</td>
<td>0.2 TB/s</td>
<td>2 TB/s</td>
<td>10 TB/s</td>
</tr>
<tr>
<td>MTTI</td>
<td>Days</td>
<td>Days</td>
<td>Days</td>
</tr>
<tr>
<td>Power</td>
<td>6 MW</td>
<td>~10 MW</td>
<td>~10 MW</td>
</tr>
</tbody>
</table>
Key Ideas to Address Memory Problem

- Memory hierarchy

- Batched data movement
  - Cache lines
  - Pages
3C’s

○ Compulsory misses
  – unavoidable misses when reading in data for the first time

○ Capacity misses
  – Not enough cache space to hold all active data

○ Conflict misses
  – Two blocks of memory are mapped into the same location in cache
Properties of Programs

○ Spatial locality
  – nearby elements will be accessed in the near future

○ Temporal locality
  – the same element will be used again in the near future
Locality discussion

We want to compute $A \times B$. The following is $A$.

What $A$ looks like in memory:
Locality discussion

We transpose $A$ to be $A'$:

$$
\begin{array}{cccc}
A1 & A2 & A3 & A4 \\
A5 & A6 & A7 & A8 \\
A9 & A10 & A11 & A12 \\
A13 & A14 & A15 & A16 \\
\end{array}
$$

What $A'$ looks like in memory:

$$
\begin{array}{cccccccccccccccc}
\end{array}
$$
3R’s

- **Rearrange (code or data)**
  - Increase spatial locality

- **Reduce size**
  - Smaller data sets easier to remain in cache

- **Reuse**
  - Increase temporal locality
Thinking more deeply

○ Given a set of data elements and an arbitrary sequence of references, can you reorganize the data elements so that the number of cache misses is minimized?

○ An NP-complete problem and you can’t find an approximate algorithm to produce results within a constant factor

○ However, finding optimal results for matrix operations is usually feasible. Why?
Case study: profile-guided code positioning

Background

- Non-trivial applications have large functions and many functions in different files

- Problematic because instruction cache performance can be poor

- Idea: rearrange basic blocks inside a function during compilation or functions during linking
Background of background

○ Basic block
  - a portion of code within a function with only one entry point and one exit point

○ Example

```
(0) L1:
(1) i := m-1
(2) j := n
(3) t1 := 4*n
(4) v := a[t1]
(5) L2: ...
```

```
(0) i := m-1
(1) j := n
(2) L4:
(3) t2 := 4 * i
(4) goto bar
(5) t3 := t3 -j
```

Mines CSCI-580, Bo Wu
Background of background

- Program instrumentation and profiling

- Instrumentation
  - Manual instrumentation
  - Compiler-based instrumentation

\[
\text{BB0:} \\
\begin{align*}
  b & := c + 1; \\
  d & := b + a; \\
  j \text{ BB2;}
\end{align*}
\]

\[
\text{BB0:} \\
\begin{align*}
  b & := c + 1; \\
  d & := b + a; \\
  \text{counter[BB0]++;} \\
  j \text{ BB2;}
\end{align*}
\]

- Profiling can be enabled by instrumentation, but can also be done through hardware-enabled monitoring
Control flow graph
- a function’s control flow graph is a directed graph, whose nodes are basic blocks

Example

```
(1) a := 0
L1:
(2) b := b+1
(3) c := c+b
(4) a := b*2
(5) if a>N goto L1
(6) return c
```

Diagram:
- BB1: $a := 0$
- BB2: $b := b+1$
- BB3: $c := c+b$
- BB4: $a := b*2$
- BB5: $id a<N goto L1$
- Return $c$
Background of background

- Function call graph
  - a directed graph representing the calling relationships

- Example

```
  X
 /|
/ 10|
 Y   Z
   2
```

Mines CSCI-580, Bo Wu
Two problems to solve

- **Procedure placement**
  - If A calls B, would like to place A and B in adjacent locations
  - Unfortunately, many functions may call B
  - This is an issue for the linker

- **Block placement**
  - Same effects occur on a smaller scale
  - If B2 always follow B1, put them together
  - This is an issue for the compiler
Ideas generated by students

- Given the basic block counters and call graph, how can you optimize the code placement?
It works

○ HP used an idea in pascal
  – Moved frequently executed blocks to top of procedure
  – 40% reduction in instruction misses
  – 5% improvement in compiler’s running time

  Wait a second. Isn’t 5% trivial?

○ Fortran compiler
  ○ Rearranged object files before linking
  ○ Attempt to improve locality on calls
  ○ 20% system throughput improvement
Procedure placement

○ Given a call graph with calling frequencies
  – Combine all edges from f_i to f_j
  – Select the highest weight edge, say x->y
    • combine x and y, along with their common edges, x->z & y->z
    • place x next to y
  – Repeat until graph cannot be reduced further
Observation:
- Targets branches with unequal execution frequencies

Would like this to become

- Unlikely path gets fall through (cheap) case
- Long distance
- In another page, ...
- Denser instruction stream

Mines CSCI-580, Bo Wu
Discussion

- Proebsting’s Law:
  - advances in compiler optimizations double computing power every 18 years
- Meaning: hardware innovation drives performance 10 times faster
- What do you think?
- Factors to consider:
  - Benchmarks (matrix multiplication, imaging processing)
  - Programming languages (JavaScript, Python, Ruby)
  - Parallelism
  - Ideas borrowed from compiler research
  - Money
Inspector-Executor for Irregular Programs
Background

Molecular Dynamics

Finite Element Analysis

Sparse Matrix Computations
Performance Drops Quickly for Irregular Applications

for(int i=0; i<N; ++i)
{
    C[i] = A[B[i]] * C[D[i]]
}

Compiler doesn’t know where the data are.
Example loop with irregular memory references

```
for i=0,7
    Y[i] = Z[x[i]]
```

- **Z** is the data array
- **x** is the index array
- Simple cache model
  - 2 elements of **Z** in each cache line
  - 2 cache lines of **Z** in cache at any one time
We can reorganize data at run-time

for $i=0,7$

\[ Y[i] = Z'[x'[i]] \]

- reorder items in data array $Z$ at run-time
- update index array $x$ with new locations
- increases spatial locality
We can also reorder iterations

```latex
\text{for } i' = 0, 7
\quad Y'[i'] = Z[x'[i']]\]
```

- reorder iterations of the i loop
- remap index array $x$ according to the new i loop ordering
- increases temporal and spatial locality
Inspector/Executor

- **Inspector**
  - Traverses through index arrays
  - Creates data and/or iteration reordering functions
  - Reorders data and updates any affected index arrays by applying reordering functions

- **Executor**
  - A transformed version of the original code which uses reordered data arrays and/or executes new iteration order
The process

- Traverses index array
- Generates data
- Reordering function $\sigma$
- Reorders data and updates index array

**Inspector**

```
for i=0,7
  ... x[i] ...

for j=0,7
  sigma[j] = ...

for j=0,7
  Z'[sigma[j]]=Z[j]
  x'[j]=sigma[x[j]]
```

**Original Code**

```
for i=0,7
  Y[i] = Z[x[i]]
```

**Executor**

```
for i=0,7
  Y[i] = Z'[x'[i]]
```
Discussion

- The executor is usually several times faster than the original loop, but do you see a problem?

```c
for(i=0; i<NITER; ++i)
{
    TransformData();
    IrregularComp();
}
```
Ideas?

○ Since we have multiple cores, how about we move the inspector to a different core and run it simultaneously with the executor?

Both inspector and executor need to modify the data, so there is data dependence.
Observations

- The transformation is composed of two parts
  - Analysis + data repositioning

```
app.  optimized data  app.
<table>
<thead>
<tr>
<th>clue   data</th>
<th>clue</th>
</tr>
</thead>
</table>
trans. | analysis  | reloc.  |
| new order |
```