Register Allocation in Kernel Generators

Matteo Frigo

Cilk Arts

July 9, 2007
Summary

- Poor register allocation $\implies$ poor kernel performance.
- Kernel generators must do register allocation one way or the other.
- Register allocation can be factored into two subproblems:
  - Scheduling.
  - Register allocation of straight-line code.
- Ordinary compilers can register-allocate straight-line code.
- Compilers cannot schedule properly. Kernel generators are responsible for the schedule.
- FFTW uses a fixed “cache oblivious” schedule. Although independent of the processor, this schedule seems to be hard to beat.
- Other problems may require more sophistication.
Impact of inefficient register allocation

32-point complex FFT in FFTW, PowerPC 7447

<table>
<thead>
<tr>
<th>add/sub</th>
<th>fma</th>
<th>load</th>
<th>store</th>
<th>code size</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>C source:</td>
<td>236</td>
<td>136</td>
<td>64</td>
<td>64</td>
<td>≈ 600 lines</td>
</tr>
</tbody>
</table>

Output of gcc-3.4 -O2:
236 136 484 285 5620 bytes  ≈ 1550

Output of gcc-3.4 -O2 -fno-schedule-insns:
236 136 134 125 2868 bytes  ≈ 640

- Twice as many instructions (all register spills).
- 2.5x slowdown.
Register allocation in gcc -O2

Assume 4 complex registers and “butterfly” instruction.
Register allocation in GCC -O2

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -02

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -02

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -O2

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -O2

Assume 4 complex registers and "butterfly" instruction.
Register allocation in gcc \texttt{-O2}

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -O2

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -02

keep going for a while...
Register allocation in gcc -O2

Assume 4 complex registers and “butterfly” instruction.
Register allocation in `gcc -O2`

Assume 4 complex registers and “butterfly” instruction.
Register allocation in gcc -O2

Assume 4 complex registers and “butterfly” instruction.
Why the gcc -O2 strategy cannot work

Theorem

If

- you compute the FFT level by level; and
- \( n \gg \) number of registers

then

- any register allocation policy incurs \( \Theta(n \log n) \) register spills.

Corollary

\( O(1) \) spills/flop no matter how many registers the machine has.
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Better strategy: blocking
Analysis of the blocking schedule

Theorem (Upper bound)
With $R$ registers,
- a schedule exists such that
- a register allocation exists such that
- the execution incurs $O(n \log n / \log R)$ register spills.

Proof.
Block for $R$ registers. Loading $R$ inputs allows you to compute \( \log R \) levels without spilling, i.e., \( \log R \) flops per spill.

Theorem (Lower bound, Hong and Kung ’81)
Any execution of the butterfly graph with $R$ registers incurs \( \Omega(n \log n / \log R) \) register spills.
Complexity of register allocation

Theorem (Motwani et al., 1995)
Given dag, find schedule of the dag and register assignment that minimizes the number of register spills: \( \text{NP-hard} \).

Theorem (Belady 1966)
Given a schedule of the dag, find register assignment that minimizes the number of register spills: \( \approx \text{linear time} \).

Corollary

- You are responsible for the schedule.
- You don't have worry about the register assignment. The compiler can do it.
Belady’s algorithm

- Traverse the schedule in execution order.
- If an instruction needs a value not in a register, obtain a register and load the value.
- If you need to evict a value from a register, evict the one used furthest in the future.
The FFTW “codelet” generator produces C.
The generator schedules the C code.
  - The scheduling algorithm is FFT-specific.
  - This is scheduling for register allocation, not “instruction scheduling” for pipelining purposes.
We assume (i.e., hope) that the C compiler implements the optimal register allocation for the given schedule.
Ordinary compilers handle straight-line code well

**Cycle counts on Pentium III, circa 2002**

<table>
<thead>
<tr>
<th></th>
<th>FFT(8)</th>
<th>FFT(16)</th>
<th>FFT(32)</th>
<th>FFT(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Belady</strong></td>
<td>150</td>
<td>350</td>
<td>838</td>
<td>2055</td>
</tr>
<tr>
<td><strong>gcc-2.95 -O2</strong></td>
<td>165</td>
<td>372</td>
<td>913</td>
<td>2254</td>
</tr>
<tr>
<td><strong>gcc-2.95 -O</strong></td>
<td>151</td>
<td>354</td>
<td>845</td>
<td>2091</td>
</tr>
<tr>
<td><strong>gcc-3.2 -O2</strong></td>
<td>152</td>
<td>390</td>
<td>892</td>
<td>2236</td>
</tr>
<tr>
<td><strong>gcc-3.2 -O</strong></td>
<td>148</td>
<td>356</td>
<td>910</td>
<td>2278</td>
</tr>
<tr>
<td><strong>icc-6.0 -O3</strong></td>
<td>166</td>
<td>397</td>
<td>946</td>
<td>2291</td>
</tr>
</tbody>
</table>
## Cycle counts on PowerPC 7400, circa 2002

<table>
<thead>
<tr>
<th></th>
<th>FFT(8)</th>
<th>FFT(16)</th>
<th>FFT(32)</th>
<th>FFT(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belady</td>
<td>112</td>
<td>272</td>
<td>688</td>
<td>1648</td>
</tr>
<tr>
<td>gcc-2.95 -O2</td>
<td>112</td>
<td>368</td>
<td>1168</td>
<td>2896</td>
</tr>
<tr>
<td>gcc-2.95 -O2 -fno-schedule-insns</td>
<td>112</td>
<td>320</td>
<td>784</td>
<td>1840</td>
</tr>
<tr>
<td>gcc-3.1 -O2</td>
<td>112</td>
<td>432</td>
<td>1312</td>
<td>3120</td>
</tr>
<tr>
<td>gcc-3.1 -O2 -fno-schedule-insns</td>
<td>112</td>
<td>288</td>
<td>768</td>
<td>1712</td>
</tr>
</tbody>
</table>
Number of spills on PowerPC 7400

<table>
<thead>
<tr>
<th></th>
<th>FFT(8)</th>
<th>FFT(16)</th>
<th>FFT(32)</th>
<th>FFT(64)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Loads:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Belady</td>
<td>5</td>
<td>21</td>
<td>75</td>
<td>146</td>
</tr>
<tr>
<td>gcc-3.1</td>
<td>6</td>
<td>26</td>
<td>107</td>
<td>251</td>
</tr>
<tr>
<td><strong>Stores:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Belady</td>
<td>5</td>
<td>21</td>
<td>64</td>
<td>133</td>
</tr>
<tr>
<td>gcc-3.1</td>
<td>6</td>
<td>23</td>
<td>73</td>
<td>155</td>
</tr>
</tbody>
</table>

(gcc-3.1 -mcpu=750 -O2 -fno-schedule-insns)
How does FFTW produce the schedule?

**Blocking:**
Could generate a different program for each $R$. (But we don’t.)

**Cache oblivious:**
It turns out that a universal schedule works well for all $R$. 
Cache oblivious FFT

Cooley-Tukey with \( p = q = \sqrt{n} \) [Vitter and Shriver]

If \( n > 1 \):

1. Recursively compute \( \sqrt{n} \) FFTs of size \( \sqrt{n} \).
2. Multiply \( O(n) \) elements by the twiddle factors.
3. Recursively compute \( \sqrt{n} \) FFTs of size \( \sqrt{n} \).

Analysis:

\[
R = \# \text{ of registers.}
\]
\[
S(n) = \# \text{ of spills in optimal register register allocation.}
\]
\[
S(n) \leq \begin{cases} 
O(n) & \text{if } n \leq \Theta(R) \\
2\sqrt{n}S(\sqrt{n}) + O(n) & \text{if } n > \Theta(R) 
\end{cases}
\]
\[
S(n) \leq O(n \log n / \log R) \quad \text{Optimal.}
\]
Cache oblivious schedule
Cache oblivious schedule
Cache oblivious schedule
Cache oblivious schedule
Machine-specific code does not seem to help

[Xiong et al., PLDI 2001]
Does this technique apply to other problems?

**Anecdotal evidence**

- **FFT, RFFT, DCT, FFT + SIMD:**
  - Butterfly-like graphs, \( O(n \log n) \) time, \( O(n \log n / \log R) \) register spills.
  - Cache oblivious works.

- **1D stencils, 1D convolutions, Gauss-Seidel, probably LCS-style “1D” dynamic programming:**
  - \( O(n^2) \) time, \( O(n^2/R) \) register spills.
  - Cache oblivious works.

- **2D stencils, GEMM/BLAS3, simple “2D” dynamic programming:**
  - \( O(n^3) \) time, \( O(n^3/\sqrt{R}) \) register spills.
  - Cache oblivious alone not sufficient. Other effects become significant.
# Matrix multiplication kernels

<table>
<thead>
<tr>
<th>Machine</th>
<th>% peak performance cache oblivious</th>
<th>% peak performance iterative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power5</td>
<td>58</td>
<td>98</td>
</tr>
<tr>
<td>UltraSPARC IIIi</td>
<td>53</td>
<td>98</td>
</tr>
<tr>
<td>Itanium II</td>
<td>93</td>
<td>94</td>
</tr>
</tbody>
</table>

[Yotov et al., 2007]
What are these “other effects”? 

- Asymptotic theory applied to small $n$ and $R$. 
- Asymmetry of loads and stores. 
- Belady does not account for the latency of spills. 
- Cache oblivious does not account for the pipeline latency.
Asymmetry of loads and stores

**Power5:**

- 2 fma/cycle.
- 2 L1 loads/cycle, in parallel with FPU.
- 1 L1 store/cycle, consumes one FPU cycle.

**Impact on** $n \times k$ by $k \times n$ **matrix multiplication kernel:**

- $2nk + n^2$ loads, $n^2$ stores.
- If cost of load is 1, cost of store is $\gamma$, then optimal aspect ratio of the kernel is nonsquare: 
  \[
  \frac{k}{n} = 1 + \gamma .
  \]
- Must modify the cache oblivious algorithm to account for $\gamma$.
- Still cache oblivious, but not $\gamma$-oblivious.
Belady and loads/stores

Theorem (Belady 1966)
Given a schedule of the dag, find register assignment that minimizes the number of loads: $\approx$ linear time.

Theorem (Farach and Liberatore 1997)
Given a schedule of the dag, find register assignment that minimizes the number of stores: NP-hard.

Theorem (Farach and Liberatore 1997)
Heuristic for the number of stores that is within a small constant factor of optimal: $\approx$ linear time. Works in practice.
Latency of reloads

**Power5:**
- FP load latency: 5 cycles.
- Must schedule 10 flops before using the loaded value.

**Problem:**
- Belady knows nothing about load latencies.

**Belady with lookahead:**
- At time $t$, schedule spills/reloads for instruction at time $t + \text{load latency}$.
- Current compilers don’t seem to do it.
- Optimal?
FPU latency

**Power5:**
- FPU latency: 6 cycles.
- 12 independent flops in flight to keep FPU busy.

**Problem:**
- Cache oblivious schedule ignores latencies.

**Possible solutions:**
- Do nothing, hope that out-of-order execution will save you.
- Attack the problem using [Blelloch and Gibbons, 2005].
Register allocation with latencies

Theorem (Blelloch and Gibbons, SPAA 2005)

Given:

- A machine with $R$ registers;
- A dag of critical path $T_{\infty}$;
- A schedule of the dag that incurs $Q_1$ spills with Belady.

Then

- A schedule of the dag exists that incurs $Q_1$ spills with at most $L T_{\infty}$ stalls on a machine with $R + L T_{\infty}$ registers and maximum latency $L$.
- The schedule is easy to compute.
- Exact result, not asymptotic.
- Optimal?
Cache oblivious DGEMM with all tricks

Power5 (peak 6.6 Gflop/s).

\((N, N) \times (N, N) \rightarrow (N, N)\) \(\forall N \in \{1, \ldots, 5000\}\).
Conclusions

This page may contain forward-looking statements that are based on management’s expectations, estimates, projections and assumptions.

- When they work, as in FFTW, universal cache oblivious kernels are attractive.
- If the “other effects” become significant, then the cache-oblivious approach is much less attractive.
- Belady/lookahead and [Blelloch and Gibbons] are kernel-independent techniques.
- Perhaps an autotuner can be structured as
  - Universal kernel-specific schedule, followed by
    - Sophisticated kernel-independent register allocator parametrized by the latencies.
- Such an autotuner would reduce the search space w.r.t. current systems.