CS 250B: Modern Computer Systems

Cache-Efficient Algorithms

Sang-Woo Jun
Back To The Matrix Multiplication Example

- Blocked matrix multiplication recap
  - C1 sub-matrix = $A_1 \times B_{11} + A_1 \times B_{21} + A_1 \times B_{31} \ldots$
  - Intuition: One full read of $B^T$ per $S$ rows in $A$. Repeated $N/S$ times

- Best performance when $S^2 \sim$ Cache size
  - Machine-dependent magic number!

\[
\begin{array}{c|c|c|c}
A & B^T & C \\
\hline
A1 & B_{11} & C1 \\
A2 & B_{12} & C2 \\
A3 & B_{13} & C3 \\
A4 & B_{14} & C4 \\
\hline
B_{21} & B_{22} & \ldots \\
B_{31} & B_{32} & \ldots \\
B_{41} & B_{42} & \ldots \\
\end{array}
\]
For sub-block size $S \times S \rightarrow N \times N \times (N/S)$ reads. What $S$ do we use?
- Optimized for L1? (32 KiB for me, who knows for who else?)
- If $S \times S$ exceeds cache, we lose performance
- If $S \times S$ is too small, we lose performance

Do we ignore the rest of the cache hierarchy?
- Say $S$ optimized for L3,
  - $S \times S$ multiplication is further divided into $T \times T$ blocks for L2 cache
- $T \times T$ multiplication is further divided into $U \times U$ blocks for L1 cache
- ...
Solution: Cache Oblivious Algorithms

- No explicit knowledge of cache architecture/structure
  - Except that one exists, and is hierarchical
  - Also, “tall cache assumption”, which is natural
- Still (mostly) cache optimal
- Typically recursive, divide-and-conquer

Tall cache assumption: \( B^2 < cM \) for a small \( c \)
(ex) Modern Intel L1: \( M: 64 \text{ KiB} \), \( B: 16 \text{ B} \)

Shorter cache with larger lines can’t efficiently divide data into small blocks
Aside: Even More Important With Storage/Network

- Latency difference becomes even larger
  - Cache: ~5 ns
  - DRAM: 100+ ns
  - Network: 10,000+ ns
  - Storage: 100,000+ ns

- Access granularity also becomes larger
  - Cache/DRAM: Cache lines (64 B)
  - Storage: Pages (4 KB – 16 KB)

Also see: “Latency Numbers Every Programmer Should Know”
https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html
Applications of Interest

- Matrix multiplication
- Merge Sort
- Stencil Computation
- Trees And Search
Cache Optimized Matrix Multiplication

How to make sure we use an optimal S, for all cache levels?
Recursive Matrix Multiplication

\[
\begin{align*}
    C_{11} & \quad C_{12} \\
    C_{21} & \quad C_{22} \\

    A_{11} & \quad A_{12} \\
    A_{21} & \quad A_{22} \\

    B_{11} & \quad B_{12} \\
    B_{21} & \quad B_{22} \\
\end{align*}
\]

\[
\begin{align*}
    C_{11} C_{12} & \quad C_{12} C_{21} \\
    C_{21} C_{22} & \quad C_{22} C_{11} \\

    A_{11} B_{11} & \quad A_{11} B_{12} \\
    A_{21} B_{11} & \quad A_{21} B_{12} \\

    A_{12} B_{21} & \quad A_{12} B_{22} \\
    A_{22} B_{21} & \quad A_{22} B_{22} \\
\end{align*}
\]

8 multiply-adds of \((n/2) \times (n/2)\) matrices
Recurse down until very small
Performance Analysis

☐ Work:
  o Recursion tree depth is $\log_2(N)$, each node fan-out is 8
  o $8^{\log_2 N} = N^{\log_2 8} = N^3$
  o Same amount of work!

☐ Cache misses:
  o Recurse tree for cache access has depth $\log(N)-1/2(\log(cM))$
    • (Because we stop recursing at $n^2 < cM$ for a small c)
  o So number of leaves = $8^{\log N-1/2 \log cM} = N^{\log 8} \div cM^{1/2 \log 8} = N^3 / cM^{3/2}$
  o At leaf, we load $cM/B$ cache lines
  o Total cache lines read = $\theta\left(\frac{n^3}{B M^{1/2}}\right)$ <- Optimal

Also, logN function call overhead is not high
Performance Oblivious to Cache Size

Steven G. Johnson, “Experiments with Cache-Oblivious Matrix Multiplication for 18.335,” MIT Applied Math
Bonus: Cache-Oblivious Matrix Transpose

- Also possible to define recursively

\[
A = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]

\[
A^T = \begin{bmatrix}
A_{11}^T & A_{21}^T \\
A_{12}^T & A_{22}^T
\end{bmatrix}
\]
Applications of Interest

- Matrix multiplication
- Trees And Search
- Merge Sort
- Stencil Computation
Trees And Search

- Binary Search Trees are cache-ineffective
  - e.g., Searching for 72 results in 3 cache line reads
  - Not to mention trees in the heap!

Each traversal pretty much hits new cache line: $\Theta(\log(N))$ cache lines read
Tree can be organized into locally encoded sub-trees

- Much better cache characteristics!
- We want cache-obliviousness:
  How to choose the size of sub-tree?
Recursive Tree Layout: van Emde Boas Layout

- Recursively organized binary tree
  - Needs to be balanced to be efficient
  - Recurses until sub-tree is size 1

- In terms of cache access
  - Recursion leaf has cache line bytes
  - Sub-tree height: $\log(B)$
  - Traverses $\log_B N$ leaf (green) trees
Performance Evaluations Against Binary Tree

1 GHz Pentium III (Coppermine)
256 KB cache
1 GB DRAM

high8, high16:
8 or 16 children per node

Brodal et al., “Cache Oblivious Search Trees via Binary Trees of Small Height,” SODA 02
Performance Evaluations Against Binary Tree And B-Tree

*High1024*: 1024 elements per node, to make use of the whole cache line (B-Tree)

Question: How do we optimize $N$ in HighN?
Databases use $N$ optimized for storage page

Note: Storage access not explicitly handled!
Letting swap handle storage management

Figure 8: Beyond main memory

Brodal et.al., “Cache Oblivious Search Trees via Binary Trees of Small Height,” SODA 02
More on the van Emde Boas Tree

- Actually a tricky data structure to do inserts/deletions
  - Tree needs to be balanced to be effective
  - van Emde Boas trees with van Emde Boas trees as leaves?
- Good thing to have, in the back of your head!
Applications of Interest

- Matrix multiplication
- Trees And Search
- Merge Sort
- Stencil Computation
Merge Sort

Depth-first  

Breadth-first

Source: https://imgur.com/gallery/voutF, created by morolin
Merge Sort Cache Effects

- Depth-first binary merge sort is relatively cache efficient
  - Log(N) full accesses on data, for blocks larger than M
  - $n \times \log\left(\frac{n}{M}\right)$

- Binary merge sort of higher fan-in (say, R) is more cache-efficient
  - Using a tournament of mergers!
  - $n \times \log_R\left(\frac{n}{M}\right)$

- Cache obliviousness: how to choose R?
  - Too large R spills merge out of cache -> Thrash -> Performance loss!
Lazy K-Merger

- Again, recursive definition of mergers!
- Each sub-merger has $k^3$ element output buffer
- Second level has $\sqrt{k} + 1$ sub-mergers
  - $\sqrt{k}$ sub-mergers feeding into 1 sub-merger
  - Each sub-merger has $\sqrt{k}$ inputs
  - $k^{3/2}$-element buffer per bottom sub-merger
  - Recurses until very small fan-in (two?)
Lazy K-Merger

Procedure Fill(v):

while v’s output buffer is not full
  if left input buffer empty
    Fill(left child of v)
  if right input buffer empty
    Fill(right child of v)
  perform one merge step

☐ Each k merger fits in $k^2$ space

☐ Ideal cache effects!
  ○ Proof too complex to show today...

☐ What should k be?
  ○ Given N elements, $k = N^{(1/3)}$ – “Funnelsort”
In-Memory Funnelsort Empirical Performance


gcc: std::sort
Funnelsort 2 vs 4:
2-way or 4-way basic merger

Improvement!
Overhead...
In-Memory Funnelsort Empirical Performance

P4 had faster memory access than Athlon. Performance bottlenecked by computation.

Source: Brodal et. al., “Engineering a Cache-Oblivious Sorting Algorithm”
In-Storage Funnelsort Empirical Performance

Source: Brodal et. al., “Engineering a Cache-Oblivious Sorting Algorithm”
Applications of Interest

- Matrix multiplication
- Trees And Search
- Merge Sort
- Stencil Computation
Stencil Computation

- Example: Heat diffusion
  - Uses parabolic partial differential equation to simulate heat diffusion

\[
\frac{\partial u}{\partial t} = \alpha \left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right)
\]
Heat Equation In Stencil Form

- Simplified model: 1-dimensional heat diffusion

\[
\frac{\partial u}{\partial t} = \lim_{\Delta t \to 0} \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}
\]

\[
\frac{\partial u}{\partial t} \approx \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t}
\]

\[
\frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} \approx k \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2}
\]

\[
\frac{\partial^2 u}{\partial x^2} = \frac{\partial u_x}{\partial x}
\]

\[
\approx \frac{u_x(x + \Delta x, t) - u_x(x, t)}{\Delta x}
\]

\[
\approx \frac{\Delta x}{u(x + \Delta x, t) - u(x, t)} - \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x}
\]

\[
\approx \frac{u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)}{(\Delta x)^2}
\]

\[
u(x, t + \Delta t) \approx u(x, t) + \alpha \left[u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)\right]
\]
A 3-point Stencil

\[ u(x, t + \Delta t) \approx u(x, t) + \alpha [u(x + \Delta x, t) - 2u(x, t) + u(x - \Delta x, t)] \]

- \( u(x, t + \Delta t) \) can be calculated using \( u(x, t) \), \( u(x + \Delta x, t) \), \( u(x - \Delta x, t) \)

- A “stencil” updates each position using surrounding values as input
  - This is a 1D 3-point stencil
  - 2D 5 point, 2D 9 point, 3D 7 point, 3D 25-point stencils popular
  - Popular for simulations, including fluid dynamics, solving linear equations and PDEs
Some Important Stencils


[2] 25-point 3D stencil for seismic wave propagation applications

Cache Behavior of Naïve Loops

- Using the 1D 3-point stencil
  - Unless $x$ is small enough, there is no cache reuse

- Continuing the theme, can we recursively process data in a cache-optimal way?
Cache Efficient Processing: Trapezoid Units

- Computation in a trapezoid is either:
  - Self-contained, does not require anything from outside ( ), or
  - Only uses data that has been computed and ready ( , after )
Recursion #1: Space Cut

- If width >= height \times 2
  - Cut the trapezoid through the center using a line of slope -1
  - Process left, then right
Recursion #2: Time Cut

- If width < height × 2
  - Cut the trapezoid with a horizontal line through the center
  - Process bottom, then top
Cache Analysis

- Intuitively, trapezoids are split until they are of size $M$ (cache size)
- Data read = $\Theta(NT/M)$
  - Cache lines read = $\Theta(NT/MB)$
  - Good!
Parallelism-Aware Cutting

- Vanilla method not good for parallelism
  - Three splits have strict dependencies...
- Space cuts can be made parallelism-friendly!
  - Bottom two first, top one next
- Effects on parallel scalability
  - Difference in impact of four cores
  - Why? DRAM bandwidth bottleneck!

<table>
<thead>
<tr>
<th>Code</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial looping</td>
<td>128.95s</td>
</tr>
<tr>
<td>Parallel looping</td>
<td>66.97s</td>
</tr>
<tr>
<td>Serial trapezoidal</td>
<td>66.76s</td>
</tr>
<tr>
<td>Parallel trapezoidal</td>
<td>16.86s</td>
</tr>
</tbody>
</table>

Performance scaling with four cores
Source: 2008-2018 by the MIT 6.172 Lecturers