Project 4 (optional) will be posted soon. 

**Topic**: Parallelizing sparse matrix vector multiplication (spmv) on GPU. 

**Due date**: 1/2/2017, 5:59pm EST. It can be a group project. Allow up to two students in a group. No late submission will be allowed.

**Extra credit**: 10% of the final course grade. Additional 5% for the group that achieves the best running time or can beat the best spmv implementation (CUSP and CUSPARSE) so far.

A list of final exam sample problems (60% of the final exam) will be posted tomorrow.
Vectorization V.S. Parallelization

Exercise: which loop is parallelizable or vectorizable?

```
do I = 1, 100
   A(I) = A(I) + 1
enddo
```

```
do I = 1, 99
   A(I) = A(I+1) + 1
enddo
```

**parallelization**

```
doall I = 1, 100
   A(I) = A(I) + 1
enddo
```

*implicit barrier sync.*

Cannot be parallelized.

**vectorization**

```
A(1:100) = A(1:100) + 1
A(1:99) = A(2:100) + 1
```
How to vectorize the following loops?

```c
for (i=2; i<100; i++) {
    S1:   a[i] = b[i+1] + 1;
    S2:   b[i] = a[i] + 5;
}
```

```c
for (i=2; i<100; i++) {
    S1:   a[i] = b[i-1] + a[i-1] + 3;
    S2:   b[i] = a[i+1] + 5;
}
```
Case Study I: A Vectorizing Source-to-Source Compiler

Assumptions

1. singly-nested loops
2. constant upper and lower bounds, step is always 1
3. body is sequence of assignment statements to array variables
4. simple array index expressions of induction variable (i +/- c or c)
5. no function calls
SKETCH OF BASIC ALGORITHM

1. Construct statement-level dependence graph considering true, anti, and output dependences.

2. Detect strongly connected components (SCC) over the dependence graph; represent SCC as summary nodes; walk resulting graph in topological order; For each visited node do
   2.1 if SCC has more than one statement in it, distribute loop around it and generate sequential code
   2.2 if SCC is a single statement and has no true dependence cycle, distribute loop around it and generate vector code; otherwise, generate sequential code
EXAMPLE

for (i=2; i<99; i++) {
    S1:  a[i] = b[i-1] + c[i-1] + 3;
    S2:  b[i] = (c[i] + b[i+1]) / 2;
    S3:  c[i] = a[i] + 1;
    S4:  d[i] = b[i] + c[i+1];
}
EXAMPLE

S2: \( b[2:99] = (c[2:99] + b[3:100]) / 2; \)
    for (i=2; i<99; i++) {
        S1: \( a[i] = b[i-1] + c[i-1] + 3; \)
        S3: \( c[i] = a[i] + 1; \)
    }

Zheng Zhang
Loop Transformation

Goal

▶ modify execution order of loop iterations
▶ preserve data dependence constraints

Motivation

▶ data locality, (increase reuse of registers, cache)
▶ parallelism, (eliminate loop-carried deps, incr granularity)

Taxonomy

▶ loop interchange, (change order of loops in nest)
▶ loop fusion, (merge bodies of adjacent loops)
▶ loop distribution, (split body of loop into adjacent loops)
▶ strip-mine and interchange (tiling, blocking), (split loop into nested loops, then interchange)
Loop Interchange

\[
\begin{align*}
\text{do } & I = 1, N \\
\text{do } & J = 1, N \\
S_1 & 
\quad A(I,J) = A(I,J-1) \\
S_2 & 
\quad B(I,J) = B(I-1,J-1) \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

\[\Rightarrow \text{loop interchange} \Rightarrow \]

\[
\begin{align*}
\text{do } & J = 1, N \\
\text{do } & I = 1, N \\
S_1 & 
\quad A(I,J) = A(I,J-1) \\
S_2 & 
\quad B(I,J) = B(I-1,J-1) \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

Loop interchange is safe \textit{iff}

\[\Rightarrow \text{it does not change the sign of the lexicographical direction vector, i.e., } (1,-1) \rightarrow (-1,1) \]

\[\Rightarrow \text{Benefits}\]

\[\circ \text{ may increase parallel loop granularity} \]
\[\circ \text{ reordering iterations may improve reuse} \]

Zheng Zhang

CS@Rutgers University
Loop Fusion

\[
\begin{align*}
\text{do } i &= 2, N \\
S_1 & \quad A(i) = B(i) \\
\text{do } i &= 2, N \\
S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

⇒ loop fusion ⇒

\[
\begin{align*}
\text{do } i &= 2, N \\
S_1 & \quad A(i) = B(i) \\
S_2 & \quad B(i) = A(i-1)
\end{align*}
\]

Loop fusion is safe \textit{iff}

- no loop-independent dependence between nests is converted to a backward loop-carried dep

(would fusion be safe if \(S_2\) referenced \(a(i+1)\) ?)

⇒ Benefits

- reduces loop overhead
- improves reuse between loop nests
- increases granularity of parallel loop
Loop distribution

$$\text{do } i = 2, N$$

$$S_1 \quad A(i) = B(i)$$

$$S_2 \quad B(i) = A(i-1)$$

⇒ loop distribution ⇒

$$\text{do } i = 2, N$$

$$S_1 \quad A(i) = B(i)$$

$$\text{do } i = 2, N$$

$$S_2 \quad B(i) = A(i-1)$$

Loop distribution is safe iff

- statements involved in a cycle of true deps (recurrence) remain in the same loop, and

- if ∃ a dependence between two statements placed in different loops, it must be forward

⇒ Benefits: necessary for vectorization, may enable partial/full parallelization, may enable other loop transformations, may reduce register/cache pressure

Zheng Zhang
Case Study II: Matrix Multiplication

Basic matrix-multiplication algorithm

for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
        Z[i,j] = 0;
        for (k = 0; k < n; k++)
            Z[i,j] = Z[i,j] + X[i,k] * Y[k,j];
    }

Computation-intensive: $3n^2$ location, $n^3$ operations

The calculations of each element of Z are independent.
Matrix Multiplication Serial Execution

\( c \) elements per cache line.

Cache misses by \( Z \) are negligible.

Cache misses by \( X \) are \( n^2 / c \).

Cache misses by \( Y \) are \( n^2 / c, n^3 / c, n^3 \).

\[
\begin{align*}
\text{for } (i=0; i < n; i++) \\
\text{for } (j=0; j < n; j++) \\
&Z[i,j] = 0.0; \\
\text{for } (k=0; k < n; k++) \\
&Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
\end{align*}
\]
Row-by-Row Parallelization

- $p$: the number of processors
- Let each processor compute $n/p$ consecutive rows of $Z$.
  - Needs to access $n/p$ rows of $X$ and $Z$, and the entire $Y$.
  - Computes $n^2/p$ elements of $Z$ with $n^3/p$ operations performed.
- The total number of cache line accesses to the caches of all processors is at least $2 \times n^2/c + p \times n^2/c$.
  - As $p$ increases, the amount of computation per core decreases, but the total communication cost increases.
Optimizations

- What are the data reuses in matrix multiply?
- Why do the reuses on X and Y result in different cache misses? (In both the serial and parallel runs.)
- A reuse yields locality (or a cache hit)
  - the reuse happens soon enough, and
  - the data is reused by the same processor.

```c
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    Z[i,j] = 0.0;
  for (k=0; k<n; k++)
    Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
```

![Diagram of matrix multiplication](image)

Zheng Zhang
Ways to Improve

- Data layout transformation
- Loop Tiling (Blocking)
Optimization I: Data Layout Transformation

Make Y memory layout column wise

- Storing Y in column-major order
- Some side effects are possible
  If Y is used in other parts of the program.
Optimization II: Blocking

▶ One way to reorder iterations in a loop.
▶ Basic idea
  ▶ Divide the matrix into submatrices (blocks)
  ▶ Order operations so an entire block is used over a short period of time.
Optimization II: Blocking

<table>
<thead>
<tr>
<th>i=0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

X

j =

0 1 ... n-1

Y

Code:

```
for (ii=0; ii < n; ii+=B)
    for (jj=0; jj < n; jj+=B)
        for (kk=0; kk < n; kk+=B)
            for (i=ii; i < ii+B; i++)
                for (j=jj; j < jj+B; j++)
                    for (k=kk; k < kk+B; k++)
                        Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
```

Performance metric:

Cache misses:
- X: $B^2/c$ per block,
totally $n^3/B^3$ blocks to bring,
hence, $n^3/Bc$ misses in total.
- Y: $n^3/Bc$ misses.

$2n^3/Bc$ misses together.

- B can be reasonably large: e.g., 200 for a 1MB cache.
- 3X speedup on an uniprocessor.