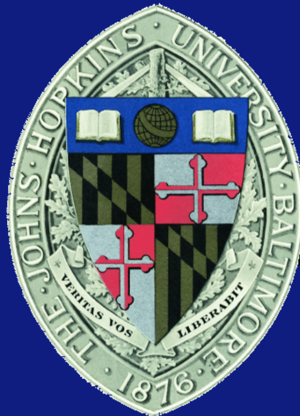
Lecture 8.2
Loop Optimizations

EN 600.320/420

Instructor: Randal Burns

22 February 2017



Department of Computer Science, *Johns Hopkins University*

How to Make Loops Faster

- Make bigger to eliminate startup costs
  - Loop unrolling
  - Loop fusion
- Get more parallelism
  - Coalesce inner and outer loops
- Improve memory access patterns
  - Access by row rather than column
  - Tile loops
- Use reductions

Loop Optimization (Fusion)

- Merge loops to create larger tasks (amortize startup)

```
// method one: distinct loops to compute A and C
for(i=0; i<N; i++){
   FFT (Npoints, A, B); // B = transformed A
   filter(Npoints, B, H); // B = B filtered with H
   invFFT(Npoints, B, A); // A = inv transformed B
}
for(i=0; i<N; i++){
   FFT (Npoints, C, B); // B = transformed C
   filter(Npoints, B, H); // B = B filtered with H
   invFFT(Npoints, B, C); // C = inv transformed B
}
```

Loop Optimization (Fusion)

- Merge loops to create larger tasks (amortize startup)

```
// method two: the above pair of loops combined into
// a single loop
for(i=0; i<N; i++){
   FFT (Npoints, A, B); // B = transformed A
   filter(Npoints, B, H); // B = B filtered with H
   invFFT(Npoints, B, A); // A = inv transformed B
   FFT (Npoints, C, B); // B = transformed C
   filter(Npoints, B, H); // B = B filtered with H
   invFFT(Npoints, B, C); // C = inv transformed B
}
```

Loop Optimization (Coalesce)

- Coalesce loops to get more UEs and thus more II-ism

```
// method one: nested loops

for(j=0; j<N; j++){
   for(i=0; i<M; i++){
        A[i][j] = work(i,j);
   }
}
```

Loop Optimization (Coalesce)

- Coalesce loops to get more UEs and thus more ll-ism

```
#pragma omp parallel for private(ij, j, i)
   for(ij=0; ij<N*M; ij++){
        j = ij/N;
        i = ij%M;

        A[i][j] = work(i,j);
   }
```

Loop Optimization (Unrolling)

- Loops that do little work have high startup costs

```
for ( int i=0; i<N; i++ )
{
  a[i] = b[i]+1;
  c[i] = a[i]+a[i-1]+b[i-1];
}
```

Loop Optimization (Unrolling)

- Unroll loops (by hand) to reduce
  - Some compiler support for this

```
for ( int i=0; i<N; i+=2 )
{
  a[i] = b[i]+1;
  c[i] = a[i]+a[i-1]+b[i-1];
  a[i+1] = b[i+1]+1;
  c[i+1] = a[i+1]+a[i]+b[i];
}
```

Memory Access Patterns

- Reason about how loops iterate over memory
  - Prefer sequential over random access (7x speedup here)
- In C, a two-dimensional array is stored in rows.

```
for (int i=0; i<n; i++)
   for (int j=0; j<n; j++)
      sum += a[i][j];
```

Figure 5.1: **Example of good memory access** – Array a is accessed along the rows. This approach ensures good performance from the memory system.

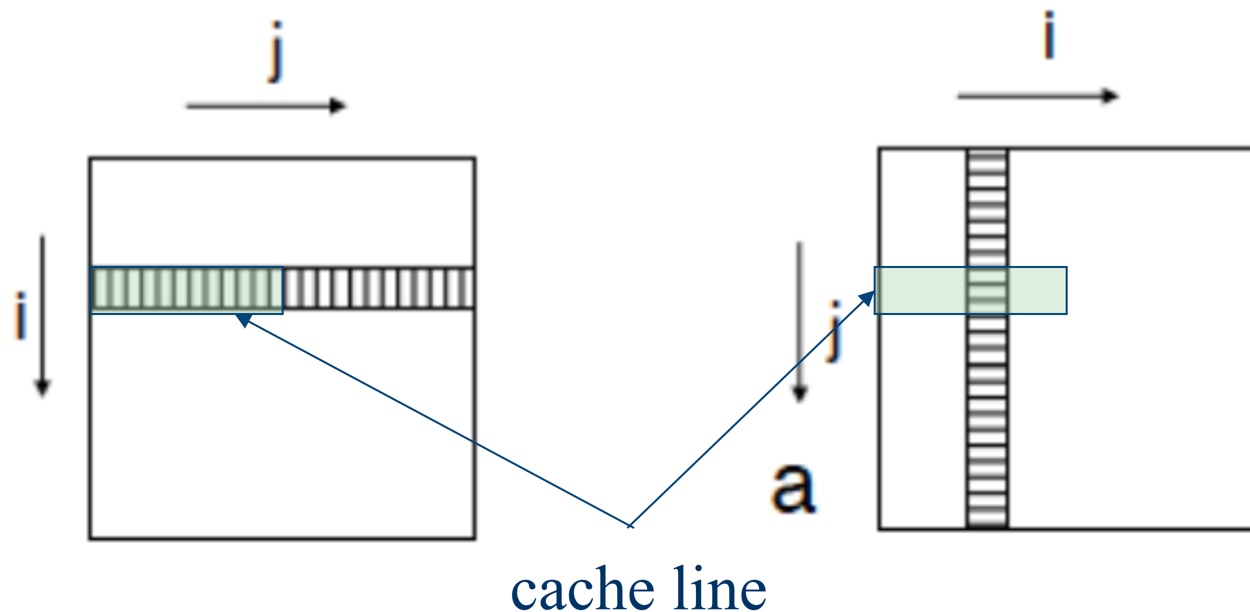
```
for (int j=0; j<n; j++)
   for (int i=0; i<n; i++)
      sum += a[i][j];
```

Figure 5.2: **Example of bad memory access** – Array a is accessed columnwise. This approach results in poor utilization of the memory system. The larger the array, the worse its performance will be.

http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf

Memory Access Patterns

- Reason about how loops iterate over memory
  - Prefer sequential over random access (7x speedup here)
- Row v. column is the classic case



http://www.akira.ruc.dk/~keld/teaching/IPDC_f10/Slides/pdf4x/4_Performance.4x.pdf

Loop Tiling

- Tiling localizes memory twice
  - In cache lines for read (sequential)
  - Into cache regions for writes (TLB hits)

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
      b[i][j] = a[j][i];
```

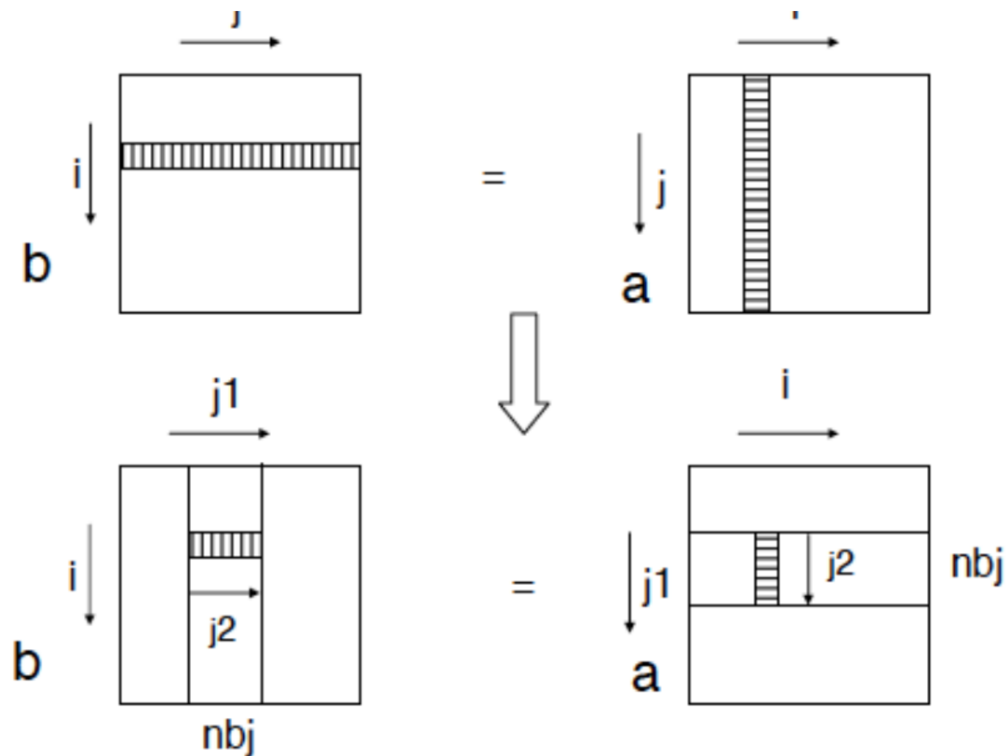
Figure 5.14: **A nested loop implementing an array transpose operation** – Loop interchange does not improve its use of cache or TLB. A fresh approach is needed.

```
for (int j1=0; j1<n; j1+=nbj)
  for (int i=0; i<n; i++)
   for (int j2=0; j2 < MIN(n-j1,nbj); j2++)
       b[i][j1+j2] = a[j1+j2][i];
```

Figure 5.15: **Loop tiling applied to matrix transpose** – Here we have used loop tiling to split the inner loop into a pair of loops. This reduces TLB and cache misses.

Loop Tiling

- Tiling localizes memory twice
  - In cache lines for write (sequential)
  - Into cache regions for writes (TLB hits)



OpenMP Reductions

- Variable sharing when computing aggregates leads to poor performance

```
#pragma omp parallel for shared(max_val)
for( i=0;i<10; i++)
{
  #pragma omp critical
  {
    if(arr[i] > max_val){
      max_val = arr[i];
    }
  }
}
```

OpenMP Reductions

- Reductions are private variables (not shared)
  - Allocated by OpenMP
- Updated by function (max) on exit for each chunk
  - Safe to write from different threads
- Eliminates interference in parallel loop

```
#pragma omp parallel for reduction(max : max_val)
for( i=0;i<10; i++)
{
  if(arr[i] > max_val){
    max_val = arr[i];
  }
}
```