HUST-USYD Summer School on Parallel Programming Practice – Lecture 2

Bing Bing Zhou (bing.zhou@sydney.edu.au)

School of Computer Science, University of Sydney

THE UNIVERSITY OF SYDNEY

Outline

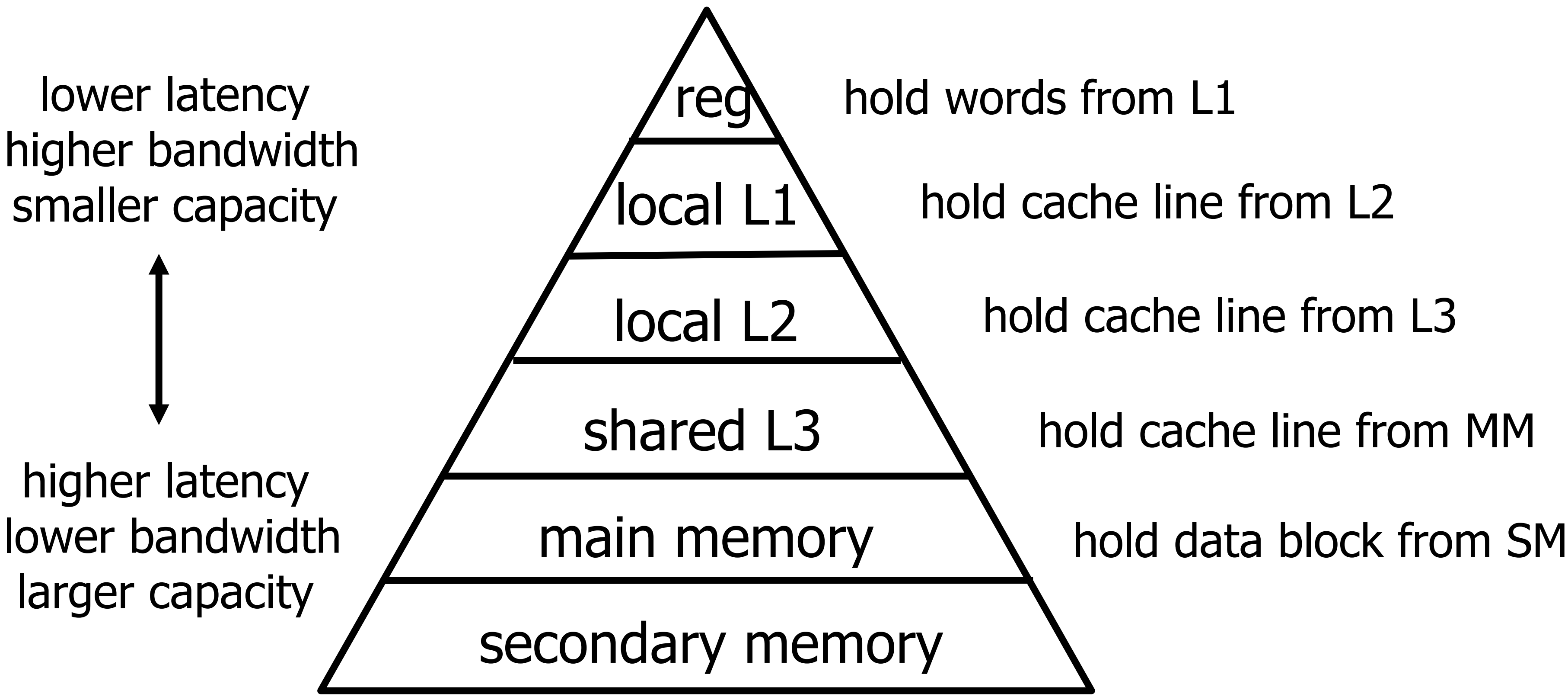
- Memory Hierarchy
- Computational intensity
- Matrix multiplication
- Contiguous memory access
- Lab exercise 1: matrix multiplication
- Blocking
- Loop unrolling
- Lab exercise 2: Matrix-vector multiplication with loop unrolling
- Homework 1: matrix multiplication with loop unrolling

Memory Hierarchy

- Most unoptimized parallel programs run at less than 10% of the machine's "peak" performance
  - Much of the performance is lost on single processors
  - Most of that loss is in the memory system
- Caches, registers and ILP are managed by hardware and compiler
  - Sometimes they do the best thing possible
  - But some other times they don't
- We need to write programs to make things more obvious to hardware and compiler for them to better optimize our codes to achieve high performance

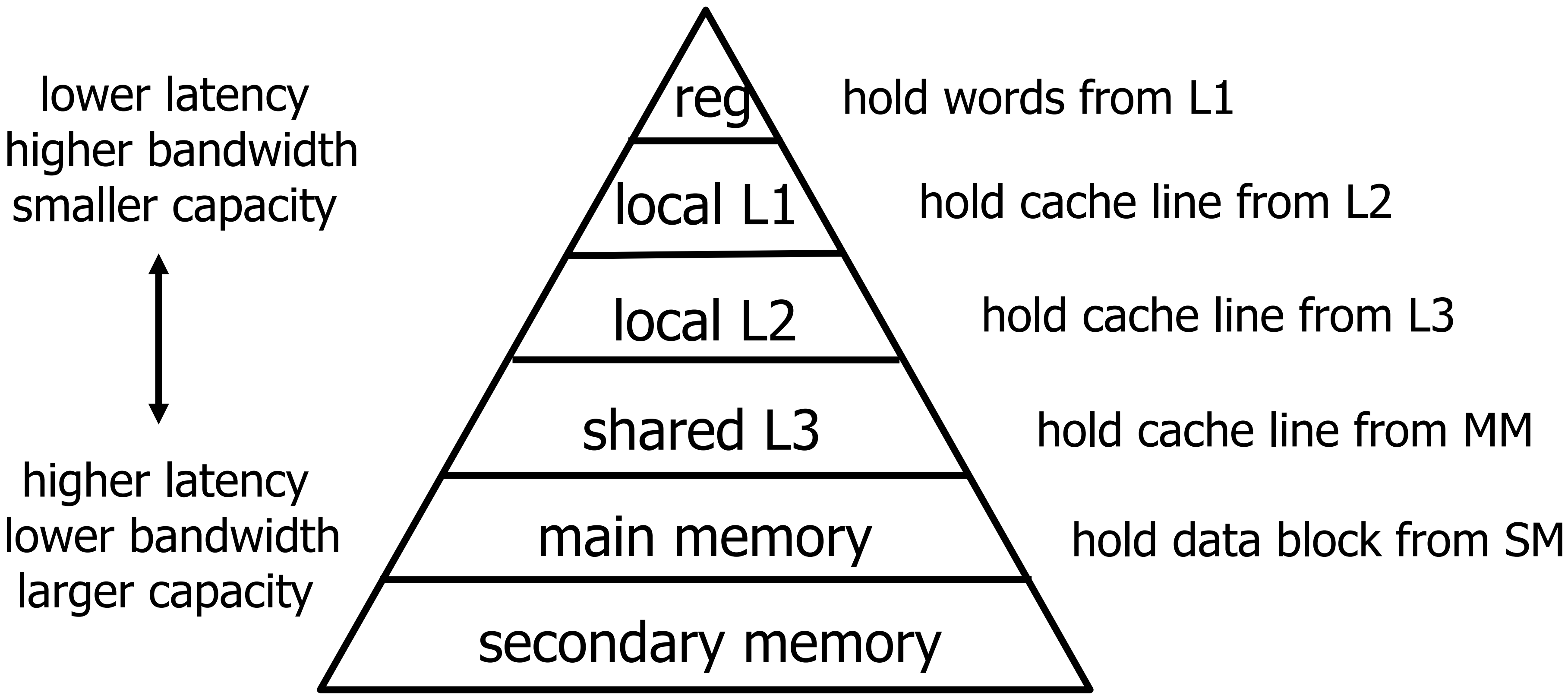
Memory Hierarchy

- In a computer memory system we have registers, caches, main memory and secondary memory
- They together form a memory hierarchy



Memory Hierarchy

- Most programs exhibit a high degree of locality
  - spatial locality: accessing items nearby previous accesses
  - temporal locality: reusing an item that was previously accessed



Memory Hierarchy

- Take advantage of memory hierarchy to improve the performance:
  - Save values in small and fast memory (cache or register) and reuse them
    - temporal locality
  - Get a chunk of contiguous data into cache (or vector register) and use whole chunk
    - spatial locality

Computational Intensity

- Assume just 2 levels in the memory hierarchy, fast and slow
- All data initially in slow memory
- Also assume:
  - m = number of memory elements (words) moved between fast and slow memory
  - t_m = time per slow memory operation
  - f = number of arithmetic operations
  - t_f = time per arithmetic operation $\ll t_m$

Computational Intensity

- Minimum possible time $= f \times t_f$ when all data in fast memory
- Actual time $= f \times t_f + m \times t_m = f \times t_f \times (1 + \frac{t_m}{t_f} \times \frac{1}{q})$
- where $q = f / m$ is the average number of flops per slow memory access – computational intensity (key to algorithm efficiency)
- Larger q means time closer to minimum $f \times t_f$

- t_m/t_f – machine balance (key to machine efficiency)
- $q \geq t_m/t_f$ needed to get at least half of peak speed

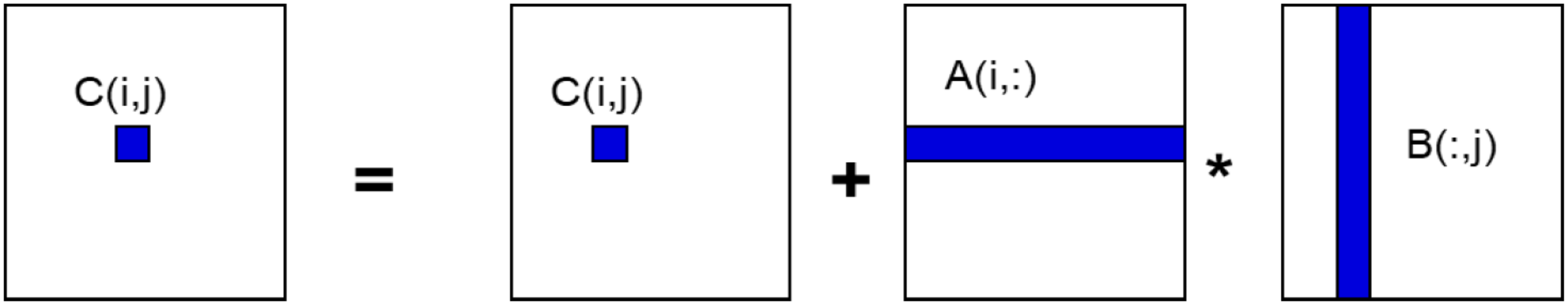
Computational Intensity

- Improve the performance on a single machine:
  - Increase computational intensity
    - Reduce cache miss rate
      - Contiguous memory access
      - Blocking
    - Make efficient use of registers
      - Loop unrolling

Matrix Multiplication

– ijk version:

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        for (k=0; k<n; k++)
            C(i, j) = C(i, j) + A(i, k) * B(k, j)
```



Matrix Multiplication

```
for (i=0; i<n; i++)
  {read row i of A into fast memory}
  for (j=0; j<n; j++)
    {read C(i, j) into fast memory}
    {read column j of B into fast memory}
    for (k=0; k<n; k++)
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
    {write C(i, j) back to slow memory}
```

Problem: fast memory too small to hold B

- Number of slow memory references:
  - $m = n^3$ read each column of B n times
  - $+ n^2$ read each row of A once
  - $+ 2n^2$ read and write each element of C twice
- $q = \frac{f}{m} = \frac{2n^3}{n^3+3n^2} \approx 2$ for large n

Matrix Multiplication

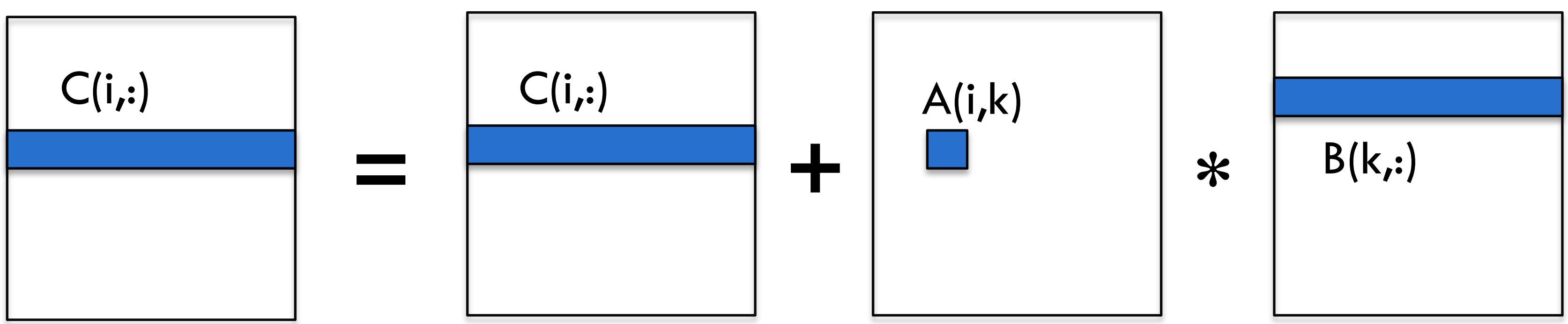
- In matrix multiplication the total number of data is $3n^2$
- Ideal q should be as large as $2n^3/4n^2 \approx O(n)$
- Thus there are rooms for improvement

- Note in matrix multiplication
  - all multiplications can be done independently
  - Addition is associative and commutative
- Then the loop orders can be changed without affecting the final multiplication results
- In the above discussion the loop order is set to ijk
- How about different loop orders
  - Consider ikj version

Contiguous Memory Access

- Data transferred between cache and main memory in cache line which consists of multiple words
- Contiguous Memory access is very important to reduce the cache miss rate and thus increase the computational intensity
- ikj version:

```
for (i=0; i<n; i++)
   for (k=0; k<n; k++)
      for (j=0; j<n; j++)
         C(i, j) = C(i, j) + A(i, k) * B(k, j)
```



Contiguous Memory Access

```
for (i=0; i<n; i++)
  for (k=0; k<n; k++)
  {read A(i, k) into fast memory}
    for (j=0; j<n; j++)
    {read row i of C into fast memory}
    {read row k of B into fast memory}
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
  {write row i of C back to slow memory}
```

- Note:
  - data loaded to cache in cache lines
  - 2D matrix placed in memory in row major order in C
  - C and B are referenced in rows
    - Assume cache line holds L words
  - Thus to access each row only need n/L slow memory accesses

Contiguous Memory Access

```
for (i=0; i<n; i++)
  for (k=0; k<n; k++)
  {read A(i, k) into fast memory}
    for (j=0; j<n; j++)
    {read row i of C into fast memory}
    {read row k of B into fast memory}
      C(i, j) = C(i, j) + A(i, k) * B(k, j)
  {write row i of C back to slow memory}
```

- Number of slow memory references:
  - $m = n^2/L$ read each element of A once
  - $+ n^3/L$ read each row of B n times
  - $+ 2n^2/L$ read and write each row of C twice
- $q = \frac{f}{m} = \frac{2n^3}{n^3/L+3n^2/L} \approx 2L$
- This is a great improvement!

Lab exercise 1: Matrix Multiplication

- Revise a simple program for matrix multiplication by changing the loop orders
- Compare the performance of different versions of matrix multiplication

Blocking Technique

- Blocking
  - Divide data into blocks for each block to fit into the cache
  - Try to use data in a block many times before the data are replaced from the cache
- Blocked Matrix Multiplication
  - Consider A,B,C (of size n-by-n) be N-by-N matrices of b-by-b subblocks where b = n / N is the block size

```
for i = 1 to N
    for j = 1 to N
      {read block C(i,j) into fast memory}
       for k = 1 to N
          {read block A(i,k) into fast memory}
          {read block B(k,j) into fast memory}
          //matrix multiply on b-by-b blocks
          C(i,j) = C(i,j) + A(i,k) * B(k,j)
       {write block C(i,j) back to slow memory}
```

Blocking Technique

– Number of slow memory references:

– $m \ = \ N * n^2$ read each block of B N^3 times ($N^3 * b^2 = N^3 * (\frac{n}{N})^2 = N * n^2$)

$+ N * n^2$ read each block of A N^3 times

$+ 2n^2$ read and write each block of C once

$= (2N + 2) * n^2$

– The computational intensity for blocked matrix multiplication:

– $q = \frac{f}{m} = \frac{2n^3}{(2N+2)n^2} \approx \frac{n}{N} = b$

– Performance can be improved by increasing the block size b>>2 (as long as $3b^2 <$ fast memory size)

Loop Unrolling

- Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program
  - Reduction of branch penalty, i.e., reduce instructions that control the loop, such as pointer arithmetic and "end of loop" tests on each iteration
  - Efficient use of multiple registers, i.e., reduce demands on memory bandwidth by pre-loading data items into registers
  - increase the computational intensity, i.e., load the data items into registers and then use many times

Convolution

```
Initialize s[i] = 0;
for (i = 0; i <= N-L; i++)
   for (j = 0; j < L; j++)
      s[i] += h[j] * a[i+j];
```

	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a
$i = 0$:	h_0	h_1	h_2	h_3					h
$i = 1$:		h_0	h_1	h_2	h_3				h
$i = 2$:			h_0	h_1	h_2	h_3			h

Convolution

```
Initialize s[i] = 0;
for (i = 0; i <= N-L; i++)
    for (j = 0; j < L; j++)
        s[i] += h[j] * a[i+j];
```

Change the loop order

```
Initialize s[i] = 0;
for (j = 0; j < L; j++)
    for (i = 0; i <= N-L; i++)
        s[i] += h[j] * a[i+j];
```

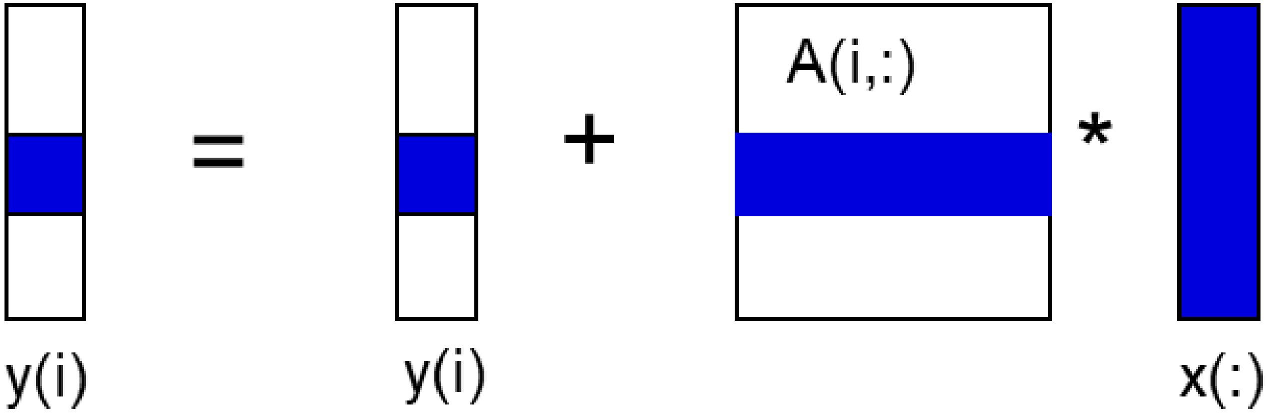
Unrolling factor = 4
Assume 4 divides L

```
Initialize s[i] = 0;
for (j = 0; j < L; j+=4){
    float h0 = h[j];
    float h1 = h[j+1];
    float h2 = h[j+2];
    float h3 = h[j+3];
    for (i = 0; i <= N-L; i++)
        s[i] += (h0 * a[i+j]
               + h1 * a[i+j+1]
               + h2 * a[i+j+2]
               + h3 * a[i+j+3]);
}
```

Matrix-Vector Multiplication

- Compute $y = y + Ax$
  - Assume matrix A is of size $n \times n$ and y and x are vectors of size n

```
for (i =0; i<n; i++)
    for (k=0; k<n; k++)
        y(i) = y(i) + a(i, k)*x(k)
```



Matrix-Vector Multiplication

– Unroll i loop with a unrolling factor = 4:

Assumed n divisible by 4
How about general cases?

```
register double b0, b1, b2, b3, x0;
for (i=0; i<n; i+=4){
  y0 = y[i]; y1 = y[i+1]; y2 = y[i+2]; y3 = y[i+3];
  for (k=0; k<n; k++) {
    x0 = x[k];
     y0 +=  a[i][k]  * x0;     y1 +=  a[i+1][k] * x0;
     y2 +=  a[i+2][k] * x0;  y3 +=  a[i+3][k] * x0;
  }
  y[i] = y0; y[i+1] = y1; y[i+2] = y2; y[i+3] = y3;
}
```

Lab exercise 2: Matrix-Vector Multiplication with Unrolling

- Revise program mv0.c to add loop unrolling with unrolling factor = 4 for general cases, i.e., n may not be divisible by 4
- Test the correctness of your program and check the performance

- After unrolling loop i, continue to unroll loop k
- Test your program for correctness and check if the performance is improved

Homework 1: Matrix Multiplication with Loop Unrolling

- Revise your programs for matrix multiplication which you did for today's lab exercise 1 by adding loop unrolling with the unrolling factor = 4
- Compare the performance by using different problem size and different number of threads
  - Draw figures or tables


Q&A