Module 2 Part 2
High Performance Application Optimization on CPUs

Carnegie Mellon University
18-645

Jike Chong
Ian Lane
With slides from Yevgen Voronenko

What we discussed last time:

Fast Platforms + **Good Techniques**

- **Multicore platforms**
- **Manycore platforms**
- **Cloud platforms**

- **Data structures**
- **Algorithms**
- **Software Architecture**

- Highlighted the difference between multicore and manycore platforms
- **Exposing** concurrency in k-means, **Exploiting** parallelism by **exploring** mappings from application to platform
- OpenMP: **An abstraction for multicore parallel programming**
- The ***choice and responsibilities*** we all have to influence the industry!

- This lecture:
  - Optimization techniques on a modern CPU
  - Mapping of a simple problem of matrix-matrix multiplication to a complex CPU

Answers you should know after this…

- Why does naïve matrix-multiply does not achieve peak performance on the CPU?
- What are the different data layouts for matrices?
- Is blocking sufficient?
- What can be learned from this for other computations?

Outline

- CPU achievable peak performance
- Matrix-multiplication discussion
- Optimization techniques

Peak Single-precision Performance

- Intel Sandy Bridge 3.5 Ghz, 4 cores * 2 sockets

[GFLOPS]	1 core	2 cores	4 cores	8 cores
None, scalar	**7**	14	28	**56**
SSE	28	56	112	224
AVX	56	112	224	**448**

- NVIDIA Tesla C2xxx
  1030 GFLOPs
- NVIDIA Quadro 5000
  722 GFLOPs

Most CPU applications achieve a fraction of this

In addition …

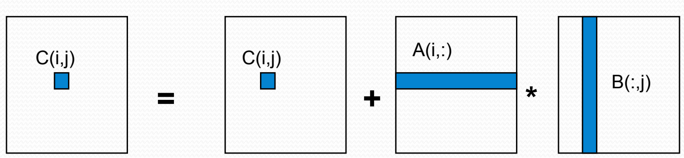
- Deep memory hierarchy
  - Registers
  - L1 cache
  - L2 cache
  - TLB (Translation Lookaside Buffer)
- Other forms of parallelism
  - Pipelining
  - Instruction-level parallelism (multiple functional units)
- "Free operations" are not free

Outline

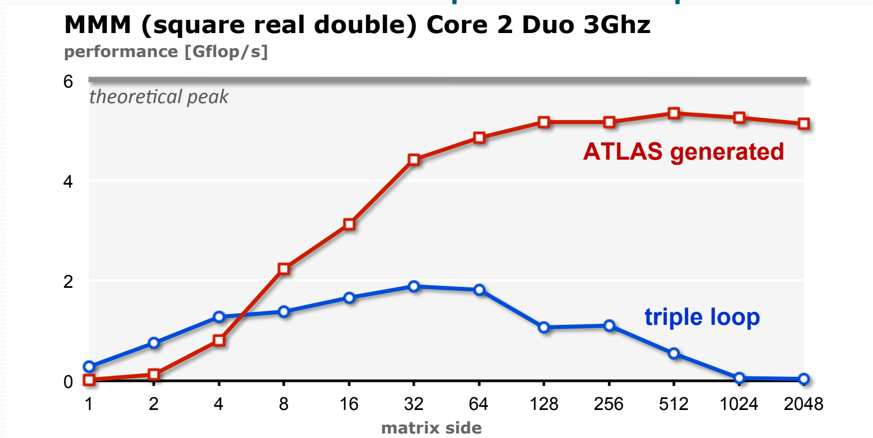
- CPU achievable peak performance
- Matrix-multiplication discussion
- Optimization techniques

Naïve Matrix Multiply

```
{implements C = C + A*B   :   Affine Transformation}
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
                C(i,j) = C(i,j) + A(i,k) * B(k,j)
```



Matrix-Matrix Multiplication Speed



- Huge performance difference for large sizes
- Great case study to learn memory hierarchy optimization

Matrix Data Layout

- A matrix is a 2-D array of elements, but memory addresses are "1-D"
- Conventions for matrix layout
  - by column, or "column major" (Fortran default);
    `A[i,j]` at `i + j*n`
  - by row, or "row major" (C default)
    `A[i,j]` at `i*n + j`

Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

Column major matrix in memory

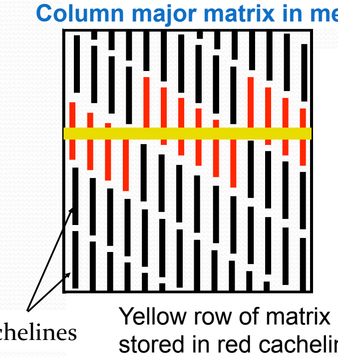
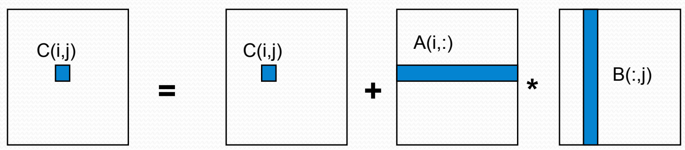


Figure source: Larry Carter, UCSD

- Column major (for now)

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
  {read row i of A into fast memory}
  for j = 1 to n
     {read C(i,j) into fast memory}
     {read column j of B into fast memory}        // O(n^3) slow loads
     for k = 1 to n
        C(i,j) = C(i,j) + A(i,k) * B(k,j)
     {write C(i,j) back to slow memory}
```



Naïve Matrix Multiply

Number of slow memory references on unblocked matrix multiply

$m = n^3$ to read each column of B n times

$+ n^2$ to read each row of A once

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

$= n^3 + 3n^2$

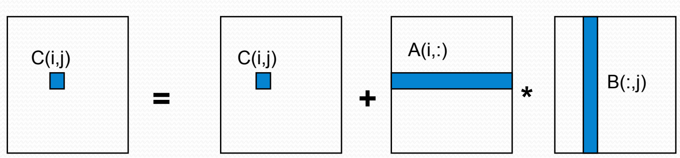
Arithmetic intensity: Total FLOPS/ Total DRAM Bytes

In single precision: $2n^3 / 4*(n^3 + 3n^2)$

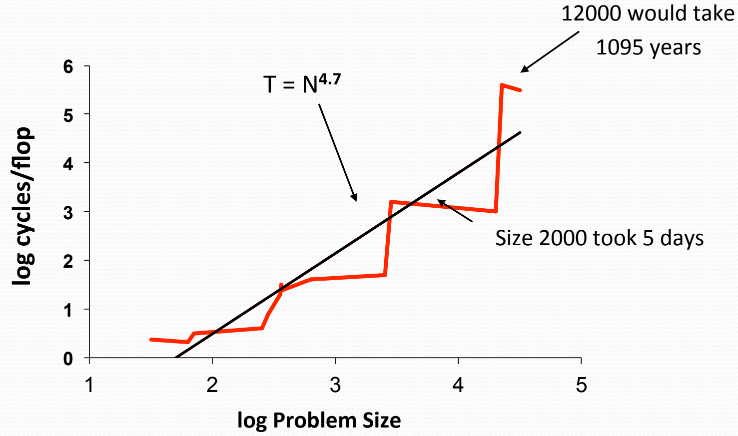
≈ 0.5 for large n, no improvement over matrix-vector multiply

Inner two loops are just matrix-vector multiply, of row i of A times B

Similar for any other order of 3 loops

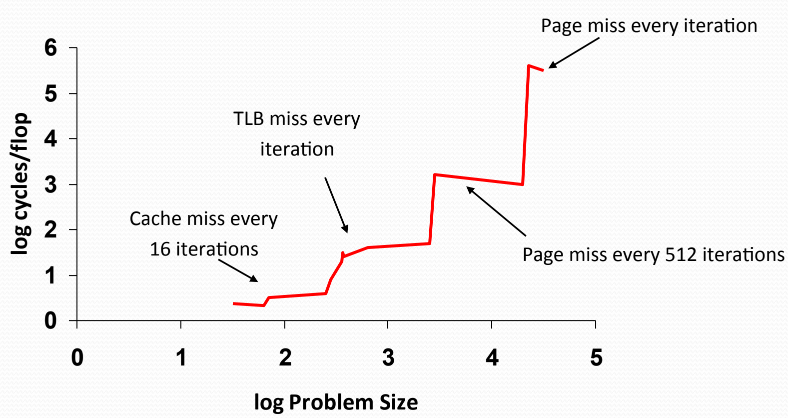


Naïve Matrix Multiply on RS/6000



O(N³) performance would have constant cycles/flop

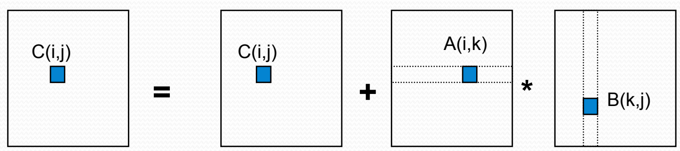
Naïve Matrix Multiply on RS/6000



Blocked (Tiled) Matrix Multiply

A,B,C = **NxN** matrices of **b x b** sub-blocks, **N = n / b**

```
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} }        // O(N^3) slow loads
      {read block B(k,j) into fast memory}
       C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
    {write block C(i,j) back to slow memory}
```



Working with a Cache Hierarchy

- Illustration by Vasily Volkov on YouTube
- Naïve Matrix Multiply:
  - http://www.youtube.com/watch?v=j5_JU5rdEi8&NR=1
- Matrix Multiply with cache blocking:
  - http://www.youtube.com/watch?v=TvelT9Bz6EU&NR=1
- Multi-level cache blocking:
  - http://stumptown.cc.gt.atl.ga.us/cse6230-hpcta-fa11/slides/11a-matmul-goto.pdf

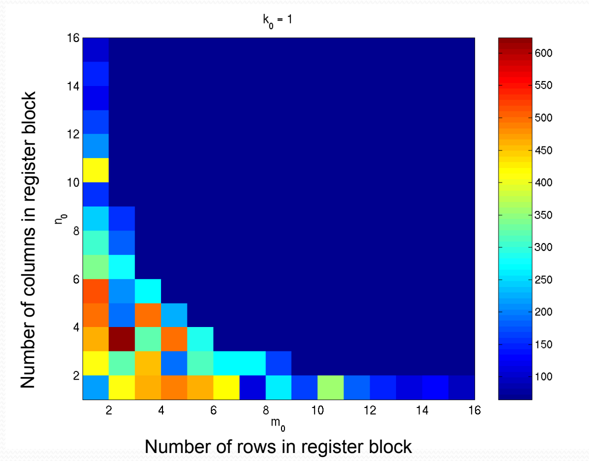
Outline

- CPU achievable peak performance
- Matrix-multiplication discussion
- Optimization techniques

Further Matrix-Multiply Optimizations in Real-world Libraries

- Block size adaptation for appropriate caches
- Register-level blocking
- Copy optimization (data layout)
- Optimizing the mini-matrix-multiply (base case)
- Multi-level blocking
- Multi-level copying

Register-level blocking



A 2-D slice of a 3-D register-tile search space. The dark blue region was pruned.

Slide source: Jim Demmel, UCB

Copy optimization

- Copy input operands or blocks
  - Reduce cache conflicts
  - Constant array offsets for fixed size blocks
  - Expose page-level locality
  - Alternative: use different data structures from start (if users willing)

Original matrix
(numbers are addresses)

0	4	8	12
1	5	9	13
2	6	10	14
3	7	11	15

Reorganized into
2x2 blocks

0	2	8	10
1	3	9	11
4	6	12	13
5	7	14	15

Slide source: Jim Demmel, UCB

Optimizing the Base Case

- Base case is usually the guts of the algorithm that computes on data in caches
- Should fit in registers
- Must use SIMD
- **Can be guided by a simple cost model**

Cost Model

- **Algebraic model**
  `+, -, /, *`
  `sin, cos, etc.`
- **Realistic model**
  - `+, -, /, *`
  - std libs (sin, cos, etc)
  - `array[i]`
  - `A.sum, A.prod`
  - `(double) x`
  - `func(…)`
  - `i < n,` compute
  - `i < n,` test and branch
  - { } unconditional branch

```
int func(int i) {
  return i+2;
}

struct A {
  double sum, prod;
};

A acc(int n, int array[]) {
  A res = {0.0, 1.0};
  for(int i=0; i<n; ++i) {
    A.sum = A.sum +
    (double)func(array[i]);
    A.prod = A.prod *
    (double)func(array[i]);
  }
  return res;
}
```

Cost Model (cont'd)

- **Algebraic model**
  `cost = 3 n`
- **Realistic model**
  ```
  + and *
  2 int -> double
  4 a.x
  ++i
  i<n conditional jump
  2 func() calls

  cost = 19 n
  ```

```
int func(int i) {
  return i+2;
}

struct A {
  double sum, prod;
};

A acc(int n) {
  A res = {0.0, 1.0};
  for(int i=0; i<n; ++i) {
    A.sum = A.sum + (double)func(i);
    A.prod = A.prod * (double)func(i);
  }
  return res;
}
```

6x discrepancy in cost

Using the Cost Model to Optimize

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc(int n, int array[]) {
  A res;
  double sum=0.0, prod=1.0;

  for(int i=0; i<n; ++i) {
    f = (double)(i+2);
    sum = sum + f;
    prod = prod * f;
  }
  return res;
}
```

Cost = 6n

Cost Reduction = closing down the gap

Using the Cost Model to Optimize II

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc(int n, int array[]) {
  A res;
  double sum=0.0, prod=1.0;

  for(int i=2; i<n+2; ++i) {
    f = (double)(i);
    sum = sum + f;
    prod = prod * f;
  }
  return res;
}
```

Cost = 5n

Optimizations enable new optimizations

Using the Cost Model to Optimize III

```
int func(int i) { return i+2; }

struct A { double sum, prod; };

A acc(int n, int array[]) {
  A res;
  double sum=0.0, prod=1.0;
  #pragma unroll(4)
  for(int i=2; i<n+2; ++i) {
    f = (double)(i);
    sum = sum + f;
    prod = prod * f;
  }
  return res;
}
```

Cost = 3.5n

Final result: 3.5n vs 19n

Cost Reduction: A Compiler Problem

- **Solution:** "human compiler"
- Compilers often fail on complex codes (many assumptions are violated)

- Optimizations
  - Strength reduction (as already shown)
  - Function inlining
  - Loop unrolling
  - Common subexpression elimination
  - Load/store elimination
  - Table lookups
  - Branch elimination

More Strength Reduction

```
for i = 1..n
    sum = sum + a[i] / c;
```

↓

```
double c_inv = 1/c;
for i = 1..n
    sum = sum + a[i] * c_inv;
```

Expensive operations:
/, %, sin, cos, log

Function Inlining

```
double f(a) { return a/c; }
...
for i = 1..n
    sum = sum + f(a[i])
```

↓

```
for i = 1..n
    sum = sum + a[i] / c;
```

Now strength reduction can apply

Loop Unrolling

```
for i = 1..n
    sum = sum + a[i]
```

↓

```
for i = 1:4:n
    sum = sum + a[i] + a[i+1] + a[i+2] + a[i+3]

// remaining iterations
for i = i:n
    sum = sum + a[i]
```

Often enables other optimizations

Common Sub-expression Elimination

```
for j = 1..m
  for i = 1..n
    sum = sum + cos(j*PI/180)*a[i];
```

↓

```
for j = 1..m
  double c = cos(j*PI/180);
  for i = 1..n
    sum = sum + c*a[i];
```

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 31

Table Lookups

```
for j = 1..m
  for i = 1..n
    sum = sum + cos(j*PI/180)*a[i];
```

↓

```
for j = 1..m
  double c = COS_TAB[j];
  for i = 1..n
    sum = sum + c*a[i];
```

18-645 – *How to Write Fast Code?* *Carnegie Mellon University (c) 2012-2014* 32

Load/Store Elimination
Loop Merging

```
for i = 1..n
    sum = sum + f(a[i]);
for i = 1..n
    prod = prod * f(a[i]);
```

```
for i = 1..n
    sum = sum + f(a[i]);
    prod = prod * f(a[i]);
```

One of the most important optimizations!
Almost always enables other optimizations.

Branch Elimination

```
for i = 1..n
   if(i != except)
      sum = sum + a[i];
```

```
for i = 1..except-1
   sum = sum + a[i];
for i = except+1..n
   sum = sum + a[i];
```

```
a[except] = 0.0;
for i = n
   sum = sum + a[i];
```

Can You Answer These Questions Now?

- Why does naïve matrix-multiply does not achieve peak performance on the CPU?
- Is blocking sufficient?
- What can be learned from this for other computations?