High-Performance Scientific Computing

Introduction

Ned Nedialkov

McMaster University

10 January 2023

Outline

Introduction

Scientific computing is about

- development of efficient numerical algorithms for solving mathematical problems that arise in science and engineering
- efficient implementation of numerical algorithms in software

Typical scenario

1. Mathematical model
2. Algorithms
3. Implementation
4. Simulation
5. Verification & Validation

High-performance computing

- is about solving (typically large) problems efficiently
- encompasses algorithms, software, and hardware

This course is about how to produce efficient implementations of numerical algorithms.

Main text: ▸ Victor Eijkhout. Introduction to High-Performance Scientific Computing

Language: C/C++

von Neuman machine

- Program and data are stored in memory.
- The processor executes each instruction in sequence
  - decode instruction
  - obtain operands from memory
  - execute the instruction
  - write the result back to memory

Modern processors

Out-of-order execution

- Processors can execute instructions
  - in a different order than specified
  - instructions are reordered if the result is not changed.

Floating-point units (FPUs)

- Can have separate addition and multiplication units.
- Fused Multiply-Add (FMA) unit
  - Can execute `a = a*x+b` as fast as addition or multiplication.
  - The (true) result is rounded only once.
- Instructions are pipelined.

Superscalar processors, Instruction-Level Parallelism (ILP)

- Analyze several instructions to find data dependencies.
- Execute instructions that do not depend on each other in parallel.

Pipelining

This loop can be pipelined

```
for (int i = 0; i < N; i++)
  x[i] = a[i] + b[i];
```

but not this one

```
for (int i = 0; i < N - 1; i++)
  x[i + 1] = a[i] * x[i] + b[i];
```

Why?

Caches

- Cache levels
  - Level 1, 2, 3 caches, L1, L2, L3.
  - L1 and L2 are part of the chip, L3 is off-chip
  - L1 is faster than L2 faster than L3

E.g. Apple M1 Pro (2022)

```
sysctl -a | grep cache | grep size
```

gives

```
hw.cachesize:  3624370176 65536 4194304 0 0 0 0 0 0 0
hw.cachelinesize:  128
hw.l1icachesize:  131072
hw.l1dcachesize:  65536
hw.l2cachesize:  4194304
```

- Cache lines
  - Cache line (cache block): how much data is moved between main memory and cache (or between caches).
  - Typically 64 or 128 bytes for L1, larger for L2.
- Data reuse
  - First time a data item is referenced it is copied from main memory into cache.
  - When referenced second time, it might be in cache.

Multicore

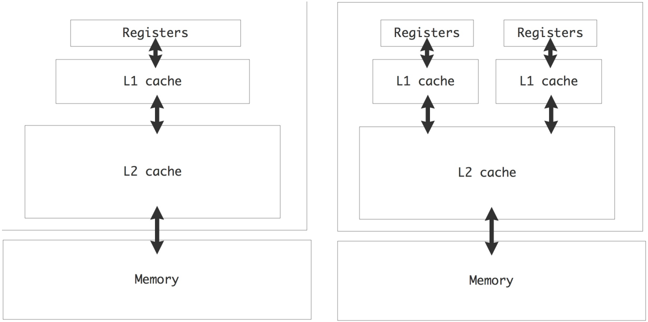

Figure: Cache hierarchy in a single-core (left) and dual-core (right) chip. Figure 1.13 from V. E. Introduction to High-Performance Scientific Computing.

Locality

Temporal locality

A data item is accessed shortly after its last access.

Spatial locality

The data items that are accessed are close to each other in memory.

Matrix multiplication

Consider multiplying two $N \times N$ matrices A and B, $C = AB$.

```
// Version 1
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
  {
    double t = 0;
    for (int k = 0; k < N; k++)
      t += A[i][k] * B[k][j];
      C[i][j] = t;
  }
```

Assume `C` contains 0's initially.

- `C[i][j]` temporal locality, can be stored in register, N^2 accesses
- `A[i][k]` accessed by rows, spatial locality, N^3 accesses
- `B[k][j]` accessed by columns, no locality, N^3 accesses

Consider

```
// Version 2
for (int i = 0; i < N; i++)
  for (int k = 0; k < N; k++)
    for (int j = 0; j < N; j++)
      C[i][j] += A[i][k] * B[k][j];
```

- `C[i][j]` spatial locality, N^3 accesses
- `A[i][k]` temporal locality, can be stored in register, N^2 accesses
- `B[k][j]` accessed by rows, spatial locality, N^3 accesses

Which version is faster?

Loop unrolling

```
// Version 1
for (int i = 0; i < N; i++)
  d += a[i] * b[i];
```

versus (assume N is multiple of 2)

```
// Version 2
for (int i = 0; i < N / 2 - 1; i++) {
  d1 += a[2 * i] * b[2 * i];
  d2 += a[2 * i + 1] * b[2 * i + 1];
}
d1 += d2;
```

What are the advantages of loop unrolling?

Read about ▸ Duff's device

Loop tiling

- Break a loop into two nested loops
- Outer loop goes over blocks of memory
- Inner loop (tile) goes through a block

Example: $A \leftarrow A + B^T$.

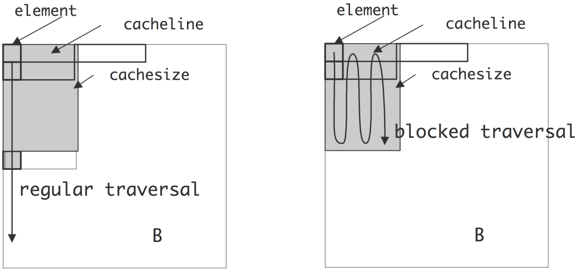

Figure: Regular and blocked traversal of a matrix.
Figure 1.25 from V. E. Introduction to High-Performance Scientific Computing.

$A \leftarrow A + B^T$

```
// Version 1
for (int i = 0; i < N; i++)
 for (int j = 0; j < N; j++)
  A[i][j] += B[j][i];
```

```
// Version 2
// bsize is block size
for (int bi = 0; bi < N; bi += bsize)
 for (int bj = 0; bj < N; bj += bsize)
  for (int i = bi * bsize; i < MIN(N, (bi + 1) * bsize); i++)
   for (int j = bj * bsize; j < MIN(N, (bj + 1) * bsize); j++)
    A[i][j] += B[j][i];
```

Compiler optimization reports

The clang compiler can produce optimization reports from the optimization passes:

`-Rpass-analysis` reports transformation analysis

`-Rpass-missed` reports missed transformations

`-Rpass` reports transformations

For example `clang -O3 -Rpass dot.c`

For more information see

- Clang Compiler User's Manual
- ‣ Compiler optimizations
- Intel's compiler. ‣ Vectorization Essentials, Vectorization and Optimization Reports