Optimizing Computations

Optimizing Computations

When the TMA methodology is applied, inefficient computations are usually reflected in the **Core Bound** and, to some extent, in the **Retiring** categories.

There are two main issues for **Core Bound** categories:

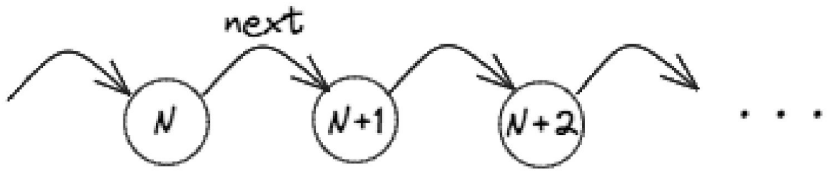
1) **Data dependencies.** For example, a long sequence of dependent operations may lead to low ILP and wasting many execution slots.

2) Shortage in hardware computing resources. This indicates that certain execution units are overloaded (also known as *execution port contention*). For example, scientific applications may run many divisions and square root operations. However, there is a limited number of multipliers and dividers in any given CPU core.

- When port contention occurs, instructions queue up waiting for their turn to be executed. This type of performance bottleneck is very specific to a particular CPU microarchitecture and usually doesn't have a cure.

Data Dependencies

- When a program statement refers to the output of a preceding statement, we say that there is a *data dependency* between the two statements.
- We also use the terms *dependency chain* or *data flow dependencies*.




```
while (n) {
  sum += n→val;
  n = n→next;
}
```

- Traversing a linked list is a typical example

Data Dependencies

- Long data dependencies forces CPU to execute code sequentially, defeating the main advantage of modern superscalar CPUs.
    - For example, pointer chasing doesn’t benefit from OOO execution and thus will run at the speed of an in-order CPU.
- Dependency chains are a major source of performance bottlenecks.
- We want to find a critical dependency chain in a hot piece of code, such as a loop or function.

Data Dependency example

Example with Random Particle Motion. *Where is the dependency chain?*

```
struct Particle {
    float x; float y; float
velocity;
};
class XorShift32 {
   uint32_t val;
   public:
   XorShift32 (uint32_t seed) :
val(seed) {}
   uint32_t gen() {
    val ^= (val << 13);
    val ^= (val >> 17);
    val ^= (val << 5);
    return val;
   }
}
```

```
/* Map degrees [0;UINT32_MAX) to radians
[0;2*pi)*/
float DEGREE_TO_RADIAN = (2 * PI_D) / UINT32_MAX;
void particleMotion(vector<Particle>
&particles,uint32_t seed) {
  XorShift32 rng(seed);
  for (int i = 0; i < STEPS; i++)
     for (auto &p : particles) {
     uint32_t angle = rng.gen();
     float angle_rad = angle * DEGREE_TO_RADIAN;
     p.x += cosine(angle_rad) * p.velocity;
     p.y += sine(angle_rad) * p.velocity;
     }
}
```

Dependency chains

Rule of thumb: if an instruction is on a critical path, look at its latency, otherwise look at its throughput.

On every loop iteration, we have 5 `fmul` and 4 `fmadd` instructions. (outside the critical path)

The `gen()` function requires 6 cc

Since the M1 processor can run 4 instructions per cycle we will have at least 9/4 = 2.25 cycles to issue all the fp instructions. So, we have two performance limits:

1) 6 cycles per iteration due to the dependency chain (SW imposed)
2) 2.25 cycles per iteration (HW imposed)

Break dependency chain

- We can employ an additional RNG object to break the chain
- Running “before” and “after” versions
  - running time go from 19ms to 10ms
  - IPC goes from 4.0 to 7.1

```
void particleMotion(vector<Particle>
&particles,uint32_t seed) {
  XorShift32 rng1(seed1);
  XorShift32 rng1(seed2);
  for (int i = 0; i < STEPS; i++)
     for (int j=0; j< particles.size(); j +=2){
      uint32_t angle1 = rng1.gen();
      uint32_t angle2 = rng2.gen();
      particles[j]   ….
      particles[j+1]   ….
     }
}
```

Inlining Functions

A function call require to pass arguments, preserve registers and manage return values.

- Usually functions uses a series of PUSH instructions (a prologue) when the function start, and a series of POP instructions (an epilogue) when the function return.
- When a function is small, the overhead of calling a function (prologue and epilogue) can be very pronounced. This overhead can be eliminated by inlining a function body into the place where it is called.
- The primary mechanism for function inlining in many compilers relies on a cost model. Inlining happens if the cost is less than a threshold, which is usually a fixed number;

Inlining Functions

Example of functions candidate for inlining

- Tiny functions (wrappers) are almost always inlined.
- Functions with a single call site are preferred candidates for inlining.

Instead, large functions usually are not inlined as they bloat the code of the caller function.

Inlining Functions

Profiling data can help in selecting functions to inline

```
Overhead | Source code & Disassembly
(%)  | of function `foo`
-----------------------------------------
3.77      : 418be0: push r15      # prologue
4.62      : 418be2: mov r15d,0x64
2.14      : 418be8: push r14
1.34      : 418bea: mov r14,rsi
3.43      : 418bed: push r13
3.08      : 418bef: mov r13,rdi
1.24      : 418bf2: push r12
1.14      : 418bf4: mov r12,rcx
3.08      : 418bf7: push rbp
3.43      : 418bf8: mov rbp,rdx
1.94      : 418bfb: push rbx
0.50      : 418bfc: sub rsp,0x8
...
# function body
...
4.17      : 418d43: add rsp,0x8  # epilogue
3.67      : 418d47: pop rbx
0.35      : 418d48: pop rbp
0.94      : 418d49: pop r12
4.72      : 418d4b: pop r13
4.12      : 418d4d: pop r14
0.00      : 418d4f: pop r15
1.59      : 418d51: ret
```

Loop Optimizations

Loops are at the heart of nearly all high-performance programs. Small changes in such a critical piece of code may have a large impact on the performance of a program.

Several techniques to improve loop performance exist:

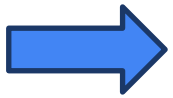
- *Low-level Optimizations:* usually apply to a single loop and can be done by compiler.
- *High-level Optimizations:* usually apply to nested loop and require must be done manually

Low-level Optimizations

Low level optimizations transform the code inside a single loop. Generally, compilers are good at doing such transformations.

- *Loop Invariant Code Motion (LICM):* when a value doesn't change across loop iterations, we can move a loop invariant expression outside of the loop.
- *Loop Unrolling:* we can unroll a loop and perform multiple iterations for each increment of the induction variable
- *Loop Strength Reduction (LSR):* replace expensive instructions with cheaper ones. Such transformation can be applied to all expressions that use an induction variable (see the sieve exercise).

```
for (int i = 0; i < N; ++i)
        a[i] = b[i * 10] * c[i];
```



```
for (int i = 0; i < N; ++i) {
        a[i] = b[j] * c[i];
        j +=10;
}
```

High-level Optimizations

High-level loop transformations change the structure of loops and often affect multiple nested loops.

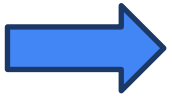
- From a compiler perspective, it is very difficult to prove the legality of such transformations and justify their performance benefit.

- Examples of optimization are:
  - Loop Interchange
  - Loop Blocking (Tiling)
  - Loop Fusion and Distribution (Fission):
  - Loop Unroll and Jam

Loop Interchange

Loop Interchange exchanges the loop order of nested loops

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



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

- Loop interchange performs sequential memory accesses to the elements of a multi-dimensional array, improving the spatial locality of memory accesses. The transformation helps to eliminate memory bandwidth and memory latency bottlenecks.
- Loop Interchange require that loops are *perfectly nested*. Interchanging imperfect loop nests require code restructuring.

Converting imperfectly-nested loops

Example 1: Isolating statements that prevent perfect loop nesting

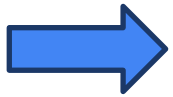
```
for (int i = 0; i < N; i++) {
   a[i]=0.0;
   for (int j = 0; j < N; j++)
        a[i] += b[i][j]*c[i][j];
}
```

```
for (int i = 0; i < N; i++)
   a[i]=0.0;
for (int j = 0; j < N; j++)
    for (int i = 0; i < N; i++)
        a[i] += b[i][j]*c [i][j];
```

Converting imperfectly-nested loops

Example 2: promoting temporary scalars to vectors

```
for (int i = 0; i < N; i++) {
    double sum = 0.0;
    double dot = 0.0;
    for (int j = 0; j < N; j++) {
          sum += b[i][j]+c[i][j];
          dot += b[i][j]*c[i][j];
    }
    a[i]= dot/sum;
}
```

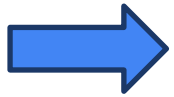


```
double sum_array [N];
double dot_array [N];
for (int i = 0; i < N; i++) {
    sum_array [i]=0.0;
    dot_array [i]=0.0;
}
for (int i = 0; i < N; i++) {
     for (int j = 0; j < N; j++) {
          sum_array[i] += b[i][j]+c[i][j];
          dot_array[i] += b[i][j]*c[i][j];
    }
}
for (int  i= 0; i < N; i++)
    a[i]= dot_array[i]/sum_array[i];
}
```

Loop Blocking (Tiling)

The idea of tiling is to split the multidimensional execution range into smaller chunks (blocks or tiles) so that each block will fit in the CPU caches.

```
// linear traversal
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        a[i][j] += b[j][i];
```



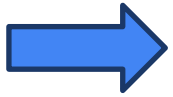
```
// traverse in 8*8 blocks
for (int ii = 0; ii < N; ii+=8)
    for (int jj = 0; jj < N; jj+=8)
        for (int i = ii; i < ii+8; i++)
            for (int j = jj; j < jj+8; j++)
                a[i][j] += b[j][i];
// remainder (not shown
```

- Typically, engineers optimize a tiled algorithm for the size of caches that are private to each CPU core. Since private caches changes from generation to generation hardcoding a block size presents its own set of challenges.

Loop Fusion and Distribution (Fission)

Separate loops can be fused when they iterate over the same range and **do not reference each other's data**.

```
for (int i = 0; i < N; i++)
     sum += b[i]+c[i];
for (int i = 0; i < N; i++)
     dot += b[i]*c[i];
```



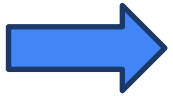
```
for (int i = 0; i < N; i++) {
     sum += b[i]+c[i];
     dot += b[i]*c[i];
}
```

- **Loop Fusion** helps to reduce the loop overhead and to improve the temporal locality of memory accesses.

Loop Fusion and Distribution (Fission)

The opposite choice is called loop distribution/fission

```
for (int i = 0; i < N; i++) {
     sum += a[i]+b[i];
     dot += c[i]*d[i];
}
```



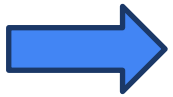
```
for (int i = 0; i < N; i++)
     sum += a[i]+b[i];
for (int i = 0; i < N; i++)
     dot += c[i]*d[i];
```

- **Loop Fission** helps in situations with a high cache contention and reduces register pressure.
- It also improve instruction cache utilization.
- Finally, when distributed, each small loop can be further optimized separately by the compiler.

Loop Unrolling

Loop unrolling increases the Instruction-Level Parallelism of the inner loop since more independent instructions are executed inside the inner loop.

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



```
for (int i = 0; i < N; i+=2) {
      diff1 += a[i]-b[i];
      diff2 += a[i+1]-b[i+1];
}
diff = diff1 + diff2;
```

- Unrolling breaks the dependency chain `diff` in the initial variant

Discovering Loop Optimization Opportunities

- Loop optimization opportunities can be identified looking into the **Compiler optimization reports** or examining the machine code.
- Many compiler **pragmas** are available to control loop transformations (loop vectorization, loop unrolling, loop distribution, etc.).
- TMA and roofline model can help to assessment loop performance. Performance is limited by one or many of the following factors:
  - memory latency,
  - memory bandwidth,
  - computing capabilities
- Once the performance bottlenecks in a loop have been identified, try applying the required transformations previously discussed.