Kattis

Constant Optimizations

Arnar Bjarni Arnarson

Árangursrík forritun og lausn verkefna

School of Computer Science

Reykjavík University

Faster I/O

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- C++: using `ios_base::sync_with_stdio(false);` at the start of your program

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- C++: using `ios_base::sync_with_stdio(false);` at the start of your program
- C++: reducing flush operations, use `'\n'` instead of `endl`

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- C++: using `ios_base::sync_with_stdio(false);` at the start of your program
- C++: reducing flush operations, use `'\n'` instead of `endl`
- C++: using a custom built function to read integers: LINK

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- C++: using `ios_base::sync_with_stdio(false);` at the start of your program
- C++: reducing flush operations, use `'\n'` instead of `endl`
- C++: using a custom built function to read integers: LINK
- Python: use `sys.stdin` and `sys.stdout`, only flush when needed. Note that `print` and `input` may flush your output, possibly slowing your program unnecessarily.

Sometimes the slowest part of your program is reading input and writing output. We can limit the impact I/O has on our program by:

- C++: using `ios_base::sync_with_stdio(false);` at the start of your program
- C++: reducing flush operations, use `'\n'` instead of `endl`
- C++: using a custom built function to read integers: LINK
- Python: use `sys.stdin` and `sys.stdout`, only flush when needed. Note that `print` and `input` may flush your output, possibly slowing your program unnecessarily.
- Java/C#: I don't know how to make it fast, if I/O is a bottleneck, use a different language maybe? Kattio.java exists but isn't particularly fast compared to C++.

Locality of Reference

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- The CPU executes instructions, which it needs to access, on data, which it also needs to access.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- The CPU executes instructions, which it needs to access, on data, which it also needs to access.
- Modern CPUs tend to separate these two and treat them differently.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- The CPU executes instructions, which it needs to access, on data, which it also needs to access.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- The CPU executes instructions, which it needs to access, on data, which it also needs to access.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.
- Temporal locality: A recently accessed (memory) address is

Locality of Reference

- Some of the optimizations discussed beyond this point may or may not be done by the compiler depending on the code you write.
- The compiler is good at optimizing common patterns, but needs a little guidance sometimes.
- We will be examining a bit how data and instructions are processed by a CPU.
- The CPU executes instructions, which it needs to access, on data, which it also needs to access.
- Modern CPUs tend to separate these two and treat them differently.
- Recently referenced, or used, data and instructions tend to be reused.
- Temporal locality: A recently accessed (memory) address is

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions
- Here array elements are accessed in increasing order with no gaps: spatial locality of data

Spotting Locality

```
int sum(int arr[N]) {
    int result = 0;
    for (int i = 0; i < N; i++) {
        result += a[i];
    }
    return result;
}
```

- Here `result` is referenced in each iteration: temporal locality of data
- Here the same instructions are used in each iteration of a short loop: temporal locality of instructions
- Here array elements are accessed in increasing order with no gaps: spatial locality of data
- Here instructions are processed sequentially: spatial locality of instructions

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.
- Swapping the order of our loops makes use of spatial locality.

Summing by columns

```
int sum_by_col(int arr[N][M]){
    int result = 0;
    for (int j = 0; j < M; j++) {
        for (int i = 0; i < N; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- In the inner loop we are jumping between memory addresses in steps of size M.
- The CPU cache will not be utilized that well, unless it is large enough to store the whole array.
- Swapping the order of our loops makes use of spatial locality.
- That way the CPU cache will be utilized better.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?
- I tested with my Intel®Core™i5-4670K

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$ to get a sufficiently large array, so 10^8 additions will be performed by each function.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$ to get a sufficiently large array, so 10^8 additions will be performed by each function.
- The function `sum_by_col` takes approximately 1.025 seconds on average.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$ to get a sufficiently large array, so 10^8 additions will be performed by each function.
- The function `sum_by_col` takes approximately 1.025 seconds on average.

Summing by rows

```
int sum_by_row(int arr[N][M]){
    int result = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; i++) {
            result += a[i][j];
        }
    }
    return result;
}
```

- Now the inner loop jumps between addresses in steps of size 1.
- So how much of a difference does this make?
- I tested with my Intel®Core™i5-4670K
- Setting $N = M = 10\,000$ to get a sufficiently large array, so 10^8 additions will be performed by each function.
- The function `sum_by_col` takes approximately 1.025 seconds on average.

A Scheduling Problem - From Errichto on CF

Problem description

There are N workers, where $1 \leq N \leq 5\,000$. Each worker is either available or unavailable each day. There is a 30 day window for a two man group project. The project can only be worked on if both group members are available. You may assume all workers are equally competent, so you only want to maximize the number of days they work together. What is the best pair of workers to select?

A Scheduling Problem - Slow Solution

```
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
```

A Scheduling Problem - Slow Solution

```
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
```

A Scheduling Problem - Slow Solution

```
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
```

A Scheduling Problem - Slow Solution

```
vector<vector<int>> workers;
int intersection(int a, int b) {
    int i = 0, j = 0;
    int result = 0;
    while (i < N && j < N) {
        if (workers[a][i] == workers[b][j]) {
            result++, i++, j++;
        }
        else if (workers[a][i] < workers[b][j]) {
            i++;
        }
        else {
            j++;
        }
    }
    return result;
```

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.

A Scheduling Problem - Less Slow Solution

```
vector<int> workers;
int intersection(int a, int b) {
    int result = 0;
    int inter = workers[a] & workers[b];
    for (int i = 0; i < D; i++) {
        if (inter & 1) {
            result++;
        }
        inter >>= 1;
    }
    return result;
}
```

- Lets store the availability as bitmasks.
- This allows us to pack our data more efficiently, which will improve our cache usage.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.
- It's time complexity is $\mathcal{O}(1)$ and most CPUs have it as a single instruction.

A Scheduling Problem - Magic Solution

```
vector<int> workers;
int intersection(int a, int b) {
    return __builtin_popcount(workers[a] & workers[b]);
}
```

- What sorcery is this?
- Popcount is the number of set bits, or ones, in the binary representation of the number.
- It's time complexity is $\mathcal{O}(1)$ and most CPUs have it as a single instruction.
- Now our time complexity is $\mathcal{O}(N^2)$ and the code runs fast enough.
- But wait, there is more!

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set.
- We have `__builtin_ffs`, which finds the index of the first set bit.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.
- What if $D = 60$? We could use 64-bit integers.

More Magic

- We have `__builtin_ctz`, which counts trailing zeros.
- We have `__builtin_clz`, which counts leading zeros.
- We have `__builtin_parity`, which returns 1 if odd number of bits are set.
- We have `__builtin_ffs`, which finds the index of the first set bit.
- For `long long` add the suffix `ll`.
- What if $D = 60$? We could use 64-bit integers.
- What if $D > 64$?

The Infamous Bitset

The Infamous Bitset

Unrolling Loops

Branching and Branchless Programming

Single Instruction, Multiple Data