TDT4200 Parallel Programming

Bart van Blokland

Lecture 3

Reference Group!

bart.van.blokland@ntnu.no

Next week (38):

Guest Lecture by Jan Christian Meyer

No lab session on Wednesday!

Week 39:

Guest lecture by Anne Elster

Lecture is moved to Thursday!

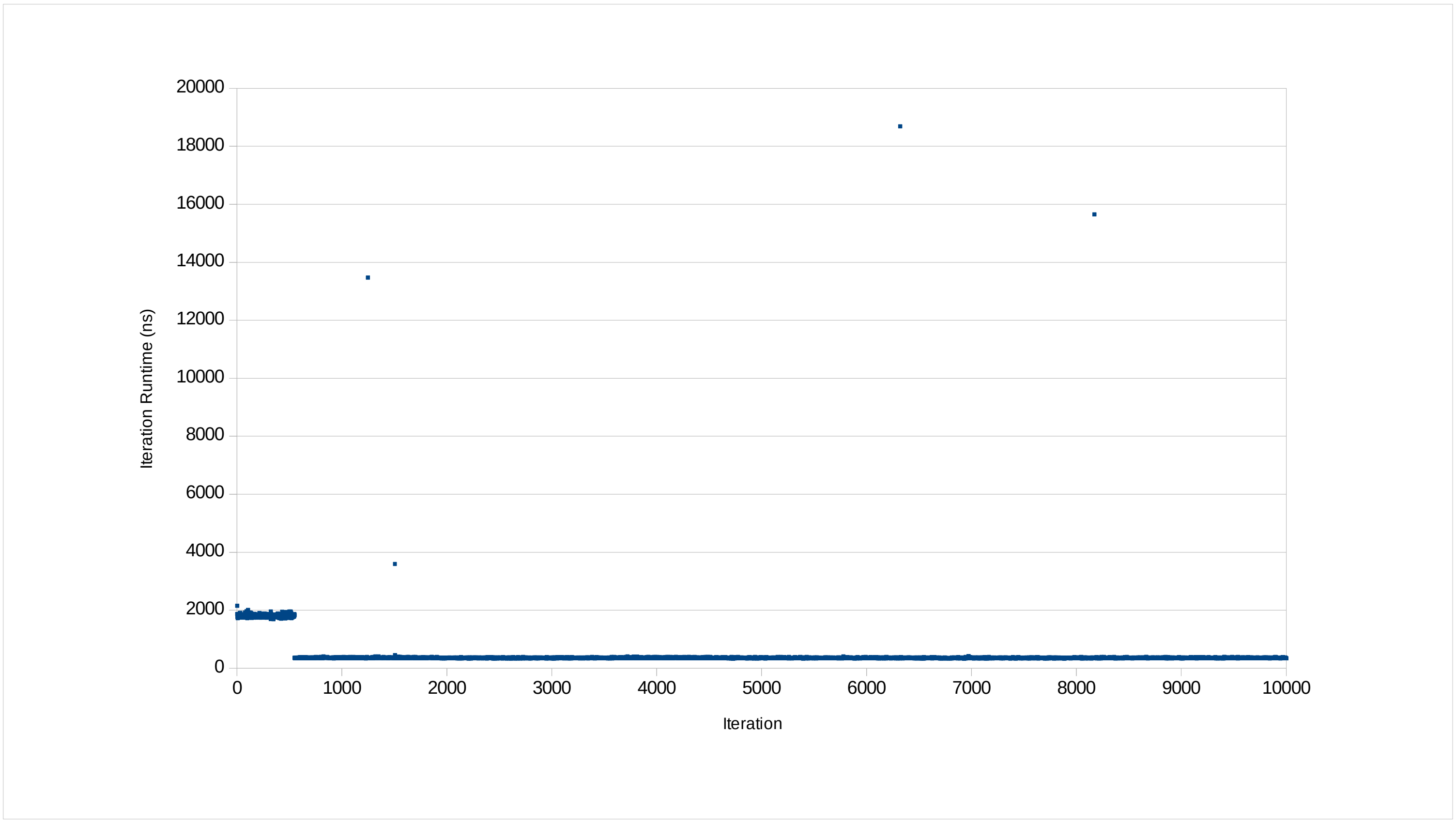
27.09, 12:00 – 14:00 in S6

Last week:

Profiling software

Measure

```cpp
#include <iostream>
#include <chrono>

int main(int argc, char** argv) {
    auto start = std::chrono::high_resolution_clock::now();

    const long iterationCount = 100000000;
    long count = 0;
    for(long i = 0; i < iterationCount; i++) {
        // I cut out the other 24 copies of this line
        count++;count++;count++;count++;count++;count++;count++;count++;count++;count++;
    }

    auto end = std::chrono::high_resolution_clock::now();

    auto timeTaken =
        std::chrono::duration_cast<std::chrono::nanoseconds>(end - start).count();

    std::cout << "Average time per iteration: "
              << double(timeTaken) / double(iterationCount) << std::endl;
}
```


20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
Iteration Runtime (ns)
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
Iteration

Internal profiler:

A profiler built into the program itself

External profiler:

An external profiling tool that hooks into the one being tested

Today:

Faste
Faster
Accelerate
ster
Boosted
Faster
Optimised
aster
More
Faster
Faster
Better
Improved
Faste
Faster
Faste

What optimisations do you not need to do?

The compiler’s job is to take your source code, and compile it into a program which has the **equivalent result** as the one in your source code.

Constant Folding

```
const int secondsPerDay = 24 * 60 * 60;
```

Constants can be precalculated by the compiler

Constant Folding

```
const int secondsPerDay = 86400;
```

Loop Unrolling

```
for(int i = 0; i < 4; i++) {
  array[i] = otherArray[i] - 10;
}
```

Loops introduce overhead by adding branch instructions and bounds checks (loop end condition). You can remove this overhead by unrolling the loop.

Loop Unrolling

```
array[0] = otherArray[0] - 10;
array[1] = otherArray[1] - 10;
array[2] = otherArray[2] - 10;
array[3] = otherArray[3] - 10;
```

Partial Loop Unrolling

```
for(int i = 0; i < 400; i++) {
  array[i] = otherArray[i] - 10;
}
```

In cases where the loop is too big to unroll, partial unrolling is also possible, where you only unroll a certain number of iterations.

Partial Loop Unrolling

```
for(int i = 0; i < 100; i+=4) {
   array[i + 0] = otherArray[i + 0] - 10;
   array[i + 1] = otherArray[i + 1] - 10;
   array[i + 2] = otherArray[i + 2] - 10;
   array[i + 3] = otherArray[i + 3] - 10;
}
```

Function Inlining

```
int addValues(int a, int b) {
   return a + b;
}

void wasteTime() {
   long sum = 0;

   for(int i = 1; i < 100; i++) {
      sum += addValues(i - 1, i);
   }
}
```

Function calls have a lot of overhead. Amongst others, the CPU needs to store its registers on the stack, needs to jump to the instructions of the other function, and needs to do the reverse when the function call is complete.

It is therefore beneficial to “inline” functions that are called a lot. You essentially copy and paste the function into the locations where it was called from. This makes the executable larger, but can give increased performance.

Function Inlining

```
void wasteTime() {
  long sum = 0;

  for(int i = 1; i < 100; i++) {
    sum += (i - 1) + i;
  }
}
```

Function Inlining

```
inline int addValues(int a, int b) {
   return a + b;
}

void wasteTime() {
   long sum = 0;

   for(int i = 1; i < 100; i++) {
      sum += addValues(i - 1, i);
   }
}
```

Function calls have a lot of overhead. Amongst others, the CPU needs to store its registers on the stack, needs to jump to the instructions of the other function, and needs to do the reverse when the function call is complete.

It is therefore beneficial to “inline” functions that are called a lot. You essentially copy and paste the function into the locations where it was called from. This makes the executable larger, but can give increased performance.

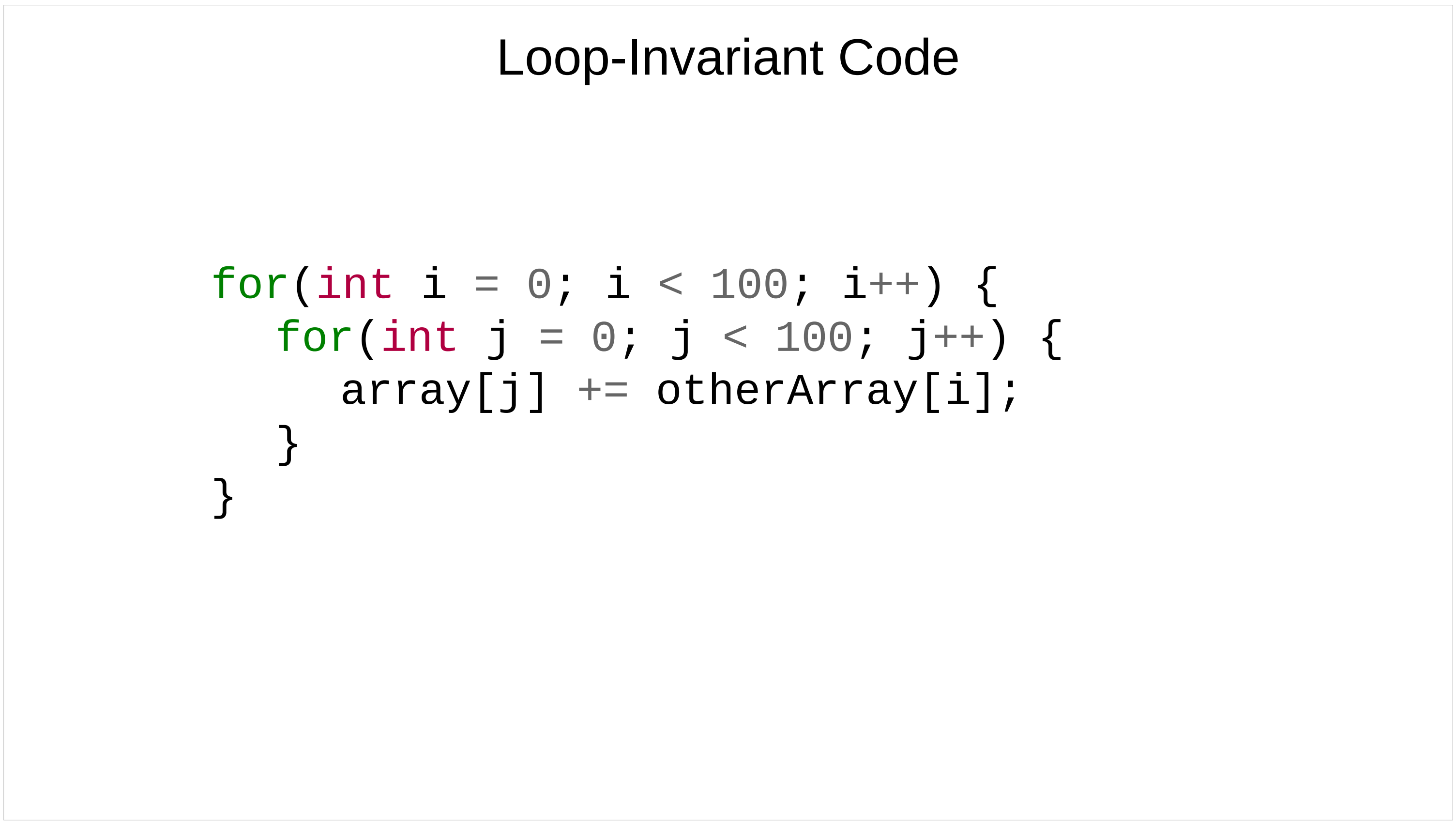
Common Subexpression Elimination

```
for(int i = 0; i < 100; i++) {
   array[4 * i + 0] = otherArray[4 * i + 0] - 10;
   array[4 * i + 1] = otherArray[4 * i + 1] - 10;
   array[4 * i + 2] = otherArray[4 * i + 2] - 10;
   array[4 * i + 3] = otherArray[4 * i + 3] - 10;
}
```

Sometimes you calculate the same value a number of times. Instead you can only calculate it once.

Common Subexpression Elimination

```
for(int i = 0; i < 100; i++) {
   int baseIndex = 4 * i;
   array[baseIndex + 0] = otherArray[baseIndex + 0] - 10;
   array[baseIndex + 1] = otherArray[baseIndex + 1] - 10;
   array[baseIndex + 2] = otherArray[baseIndex + 2] - 10;
   array[baseIndex + 3] = otherArray[baseIndex + 3] - 10;
}
```



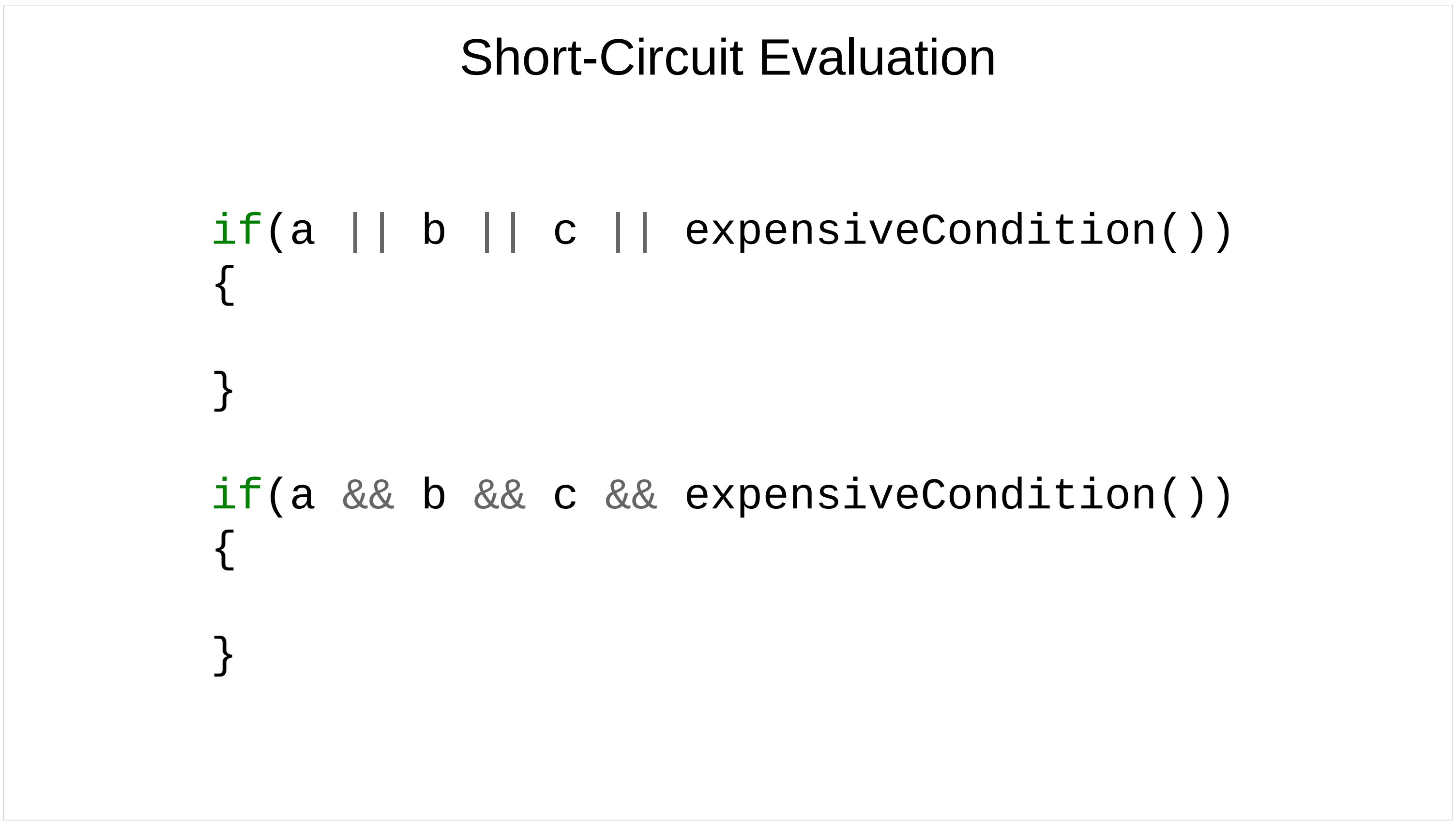
Loop-Invariant Code

```
for(int i = 0; i < 100; i++) {
  for(int j = 0; j < 100; j++) {
    array[j] += otherArray[i];
  }
}
```

Some code within a loop does not change. You can move it outside the loop instead, and reuse the result.

Loop-Invariant Code

```
for(int i = 0; i < 100; i++) {
  int otherValue = otherArray[i];
  for(int j = 0; j < 100; j++) {
    array[j] += otherValue;
  }
}
```



Short-Circuit Evaluation

```cpp
if(a || b || c || expensiveCondition())
{

}

if(a && b && c && expensiveCondition())
{

}
```

Something which the compiler is always forced to do by the C++ standard is short-circuit evaluation. When evaluating boolean expressions like the ones above, if a is <u>true</u> in the above statement, and <u>false</u> in the bottom one, there is no point in evaluating the remainder. The result will be the same regardless. It’s therefore useful to try and order the conditions by how expensive they are to compute.

Strength Reduction

Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957

Strength Reduction

```
for(int i = 0; i < 100; i++) {
  minutes[i] = float(seconds[i]) / 60.0f;
}
```

Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957

Strength Reduction

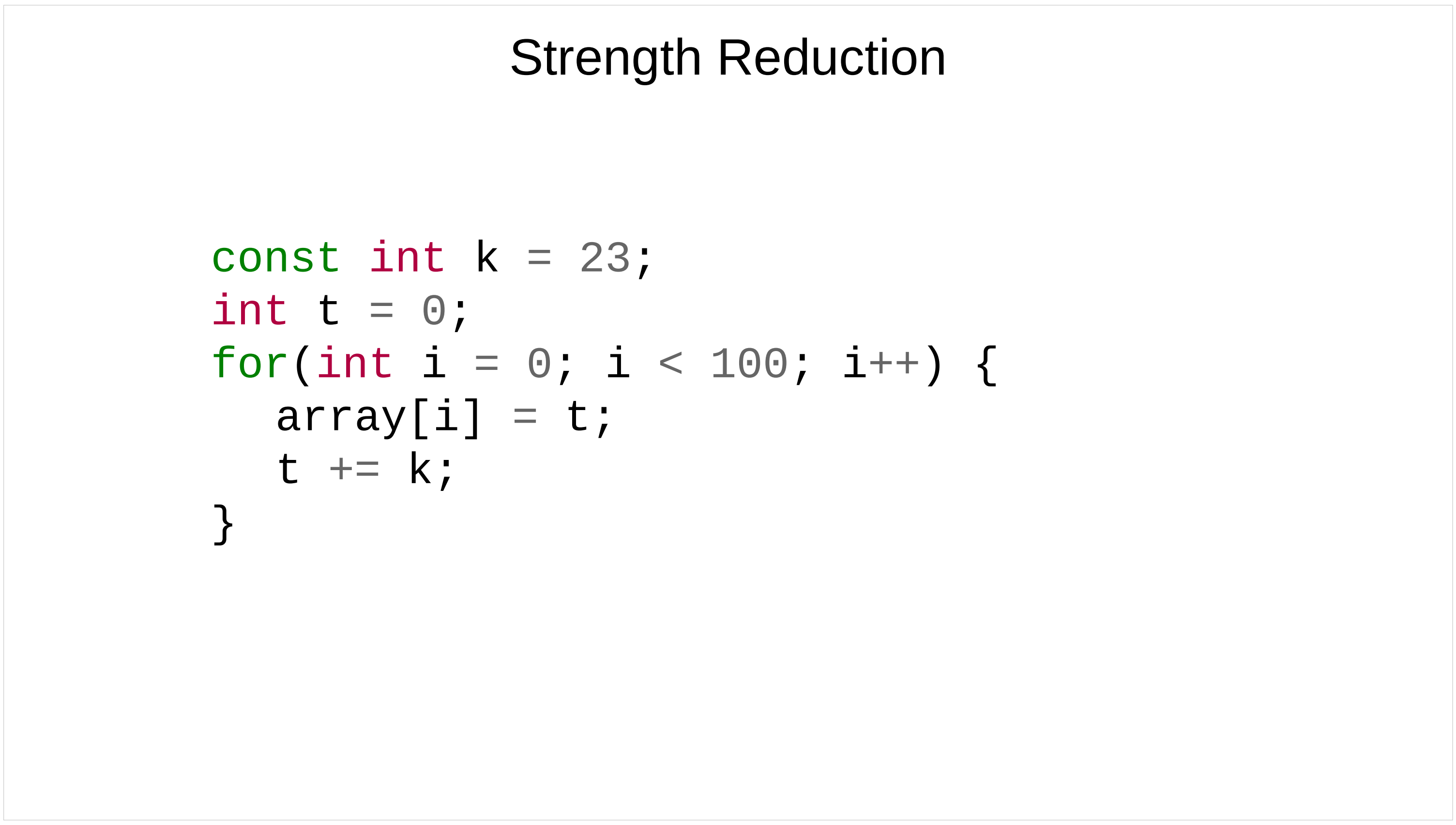
```
const float fraction = 1.0 / 60.0;
for(int i = 0; i < 100; i++) {
  minutes[i] = float(seconds[i]) * fraction;
}
```

Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957

Strength Reduction

```
const int k = 23;
for(int i = 0; i < 100; i++) {
  array[i] = i * k
}
```

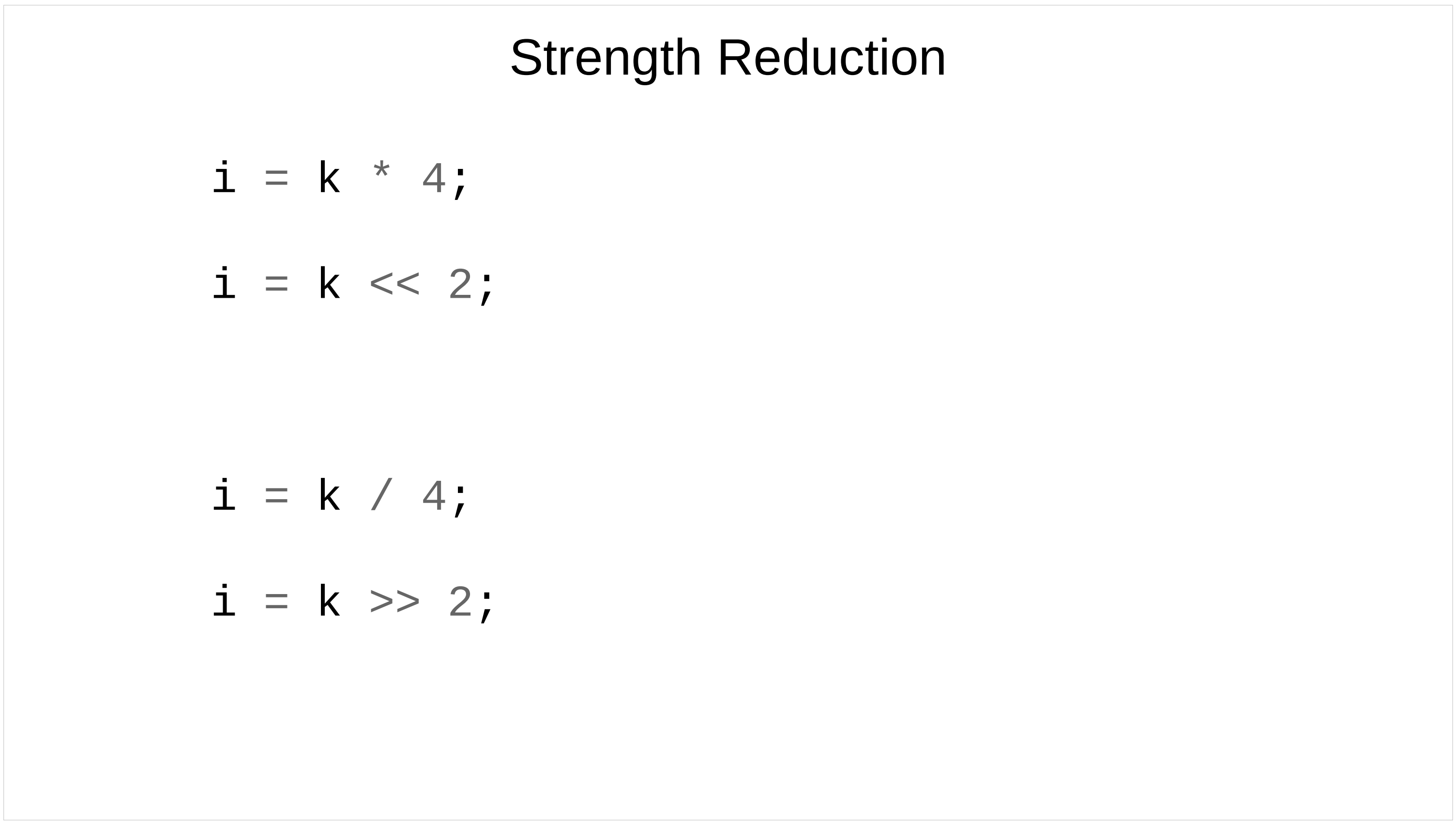
Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957



Strength Reduction

```
const int k = 23;
int t = 0;
for(int i = 0; i < 100; i++) {
   array[i] = t;
   t += k;
}
```

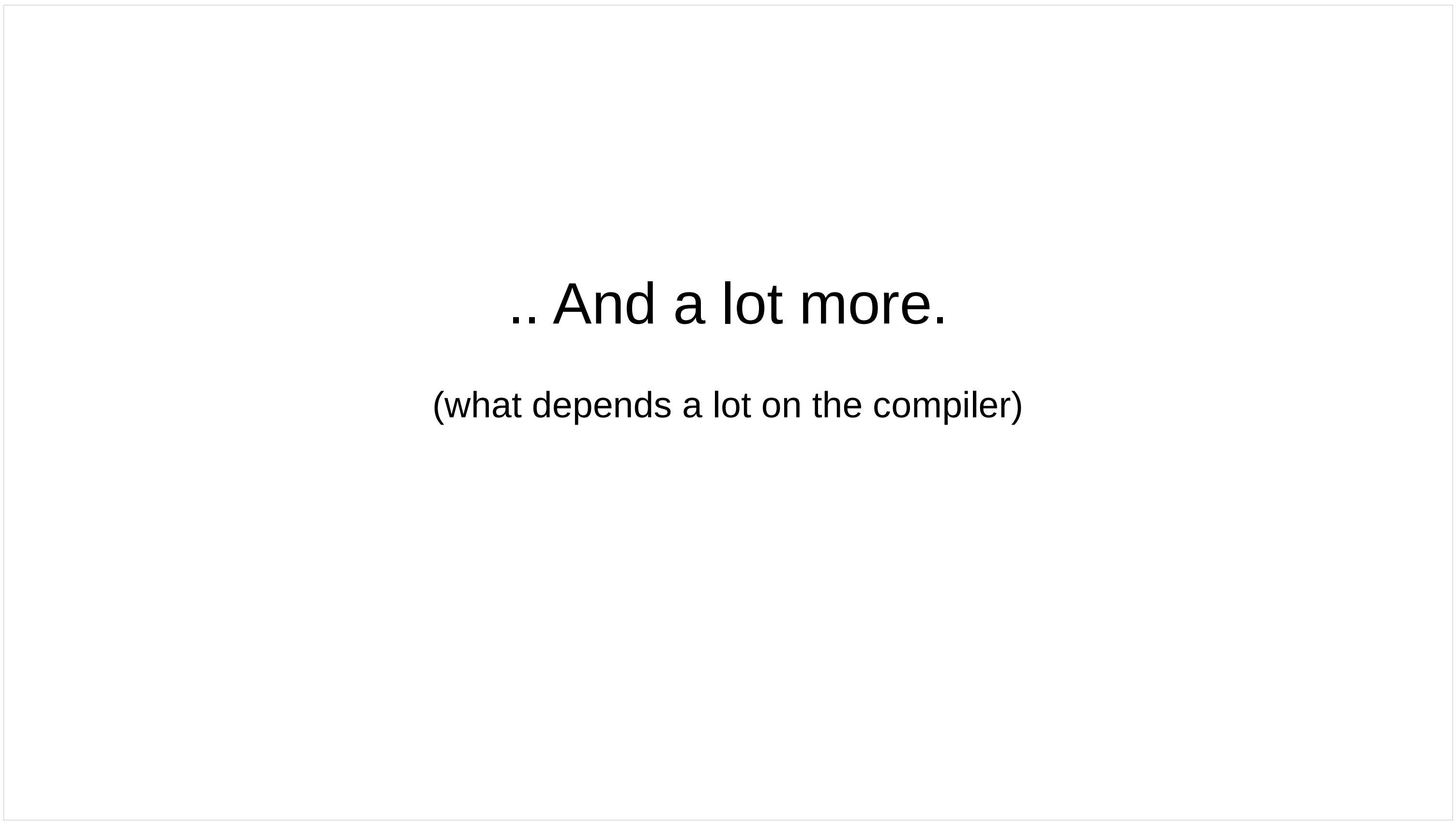
Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957



Strength Reduction

```
i = k * 4;

i = k << 2;



i = k / 4;

i = k >> 2;
```

Image source: https://tenor.com/view/star-wars-obi-wan-kenobi-lightsaber-weak-gif-3497957



.. And a lot more.

(what depends a lot on the compiler)

A lot has to do with how to best map the program on to the target hardware.

One example is reordering statements, and performing analysis on where variables are used.

But there are limits.

Compilers will not always apply optimisations.

Example: Maintain floating point accuracy

A lot has to do with how to best map the program on to the target hardware.

Compilers are not omniscient. They can take your code and improve it by quite a bit, but the complexity involved in doing so is immense already.

In addition, some optimisations require making particular assumptions which the C++ language does not always allow the compiler to make. This can give you some nasty surprises.

What can YOU do?

Image credit: https://www.pinterest.com/pin/463307880406845051/

Use a better compiler

Use a better algorithm

Use a better library

A compiler is a tool, not a .

Use a better compiler

Use a better algorithm

Use a better library

Rule of thumb:

Better algorithm: 30x - 100x+ speedup

Optimisation: < ~10x

Please note:

Focus on bottlenecks

These tips are not always applicable

Optimisation is a skill, not a cookbook

In general:

1. Minimise heap allocations

2. The CPU cache is your friend

3. Remove computation

(4. Use parallelism)

Minimise heap allocations

`new` is expensive.

(typically 1000s of instructions)

Allocate memory in large chunks

Reuse memory

Example: Keep std::vectors between iterations

Reduce memory accesses

(if your cache can handle it)

Help the CPU hardware

Use a struct of arrays, not an array of structs

```
struct float3 {
  float x, y, z;
};

struct Vertex {
  float3 position;
  float3 normal;
  float3 colour;
}

void doSomething() {
  Vertex* vertices = new Vertex[100000];

  for(int i = 0; i < 100000; i++) {
    vertices[i].position.x += 5;
  }
}
```

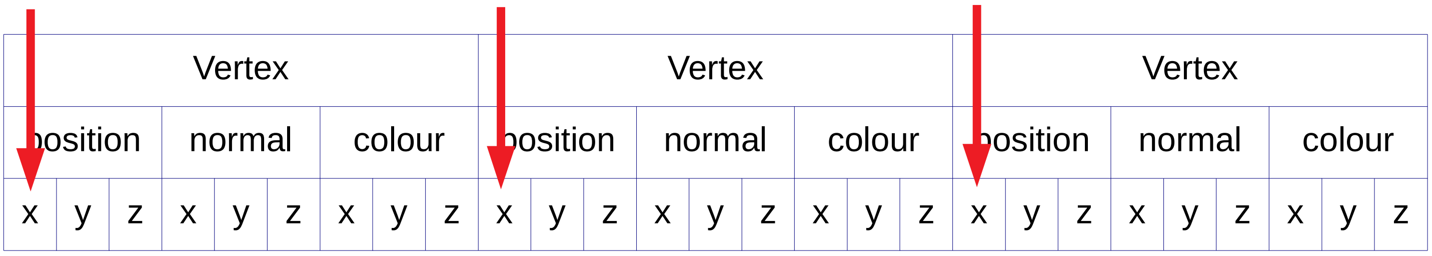
Use a struct of arrays, not an array of structs

```
struct float3 {
  float x, y, z;
};

struct Vertex {
  float3 position;
  float3 normal;
  float3 colour;
}

Vertex* vertices = new Vertex[100000];
```

Vertex									Vertex									Vertex								
position			normal			colour			position			normal			colour			position			normal			colour		
x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z

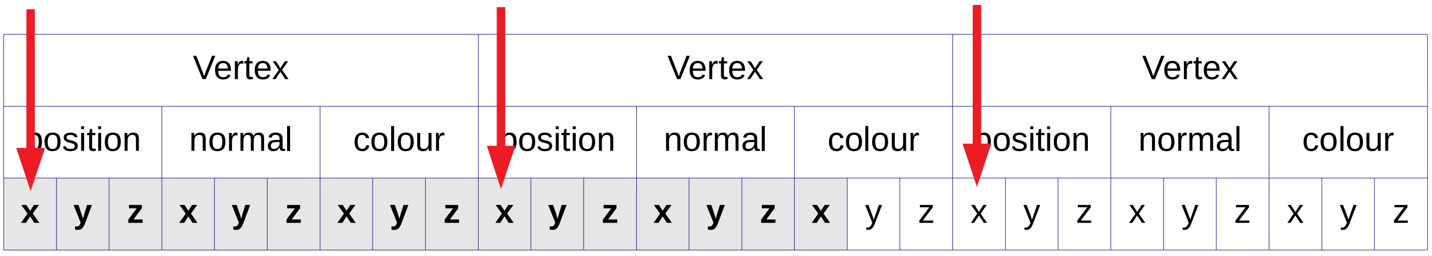
Use a struct of arrays, not an array of structs

```
struct float3 {
   float x, y, z;
};

struct Vertex {
   float3 position;
   float3 normal;
   float3 colour;
}

Vertex* vertices = new Vertex[100000];
```


Vertex | Vertex | Vertex
position | normal | colour | position | normal | colour | position | normal | colour
x | y | z | x | y | z | x | y | z | x | y | z | x | y | z | x | y | z | x | y | z | x | y | z | x | y | z

Use a struct of arrays, not an array of structs

```
struct float3 {
   float x, y, z;
};

struct Vertex {
   float3 position;
   float3 normal;
   float3 colour;
}

Vertex* vertices = new Vertex[100000];
```

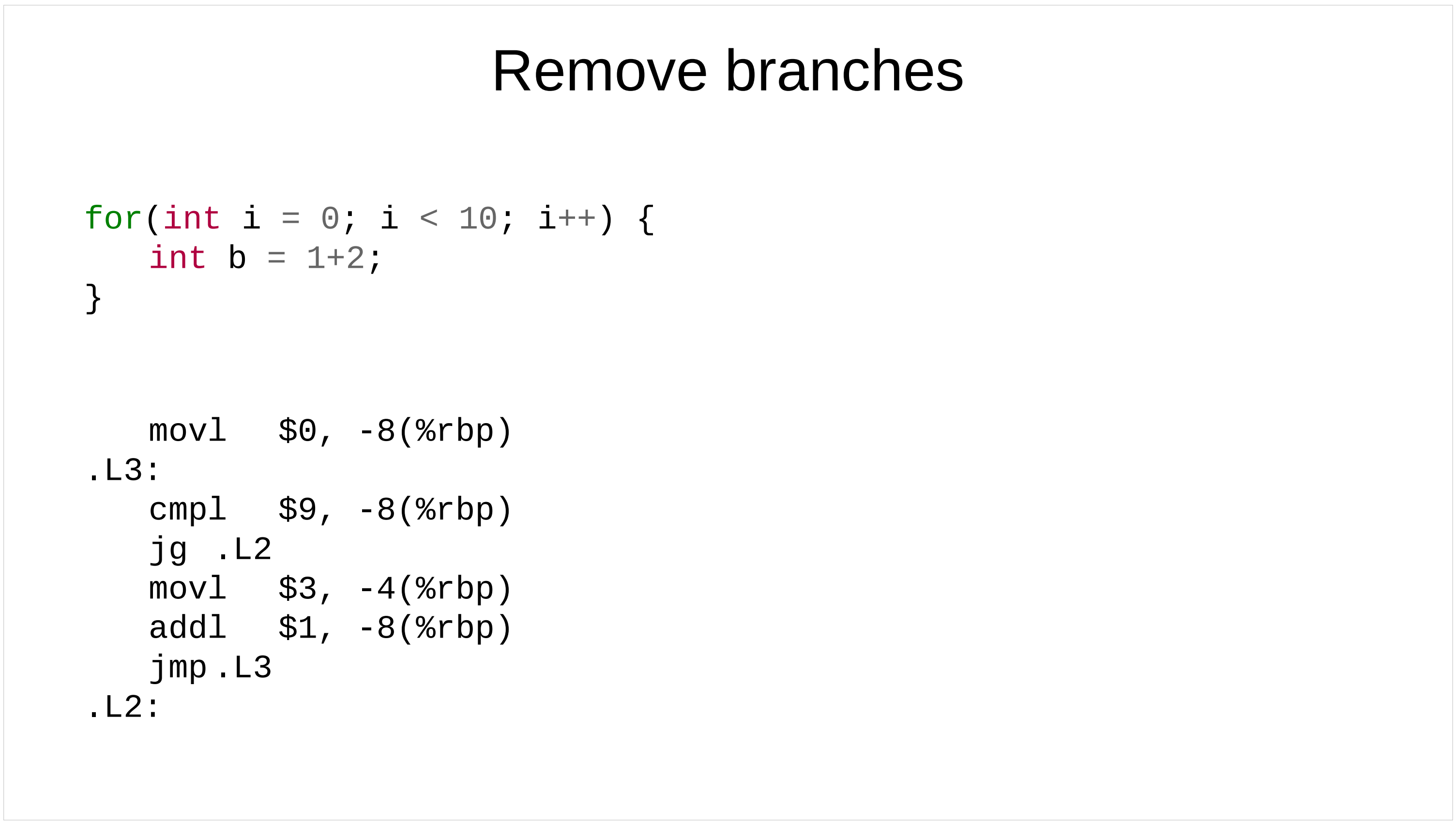


Use a struct of arrays, not an array of structs

```
struct Vertices {
  float* position_x;
  float* position_y;
  float* position_z;
  float* normal_x;
  float* normal_y;
  float* normal_z;
  float* colour_x;
  float* colour_y;
  float* colour_z;
}

Vertices verts;
verts.position_x = new float[100000];
verts.position_y = new float[100000];
verts.position_z = new float[100000];
verts.normal_x = new float[100000];
```

Remove branches

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg



Remove branches

```
for(int i = 0; i < 10; i++) {
   int b = 1+2;
}
```

```
   movl   $0, -8(%rbp)
.L3:
   cmpl   $9, -8(%rbp)
   jg  .L2
   movl   $3, -4(%rbp)
   addl   $1, -8(%rbp)
   jmp .L3
.L2:
```

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove branches

```
for(int i = 0; i < 10; i++) {
    int b = 1+2;
}
```

```
    set reg0 to 0
label L3
    compare reg0 to 9
    jump if greater to L2
    set reg1 to 3
    add 1 to reg0
    jump to L3
label L2
```

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove branches

```
for(int i = 0; i < 10; i++) {
   int b = 1+2;
}
```

```
   set reg0 to 0
label L3
   compare reg0 to 9
   jump if greater to L2
   set reg1 to 3
   add 1 to reg0
   jump to L3
label L2
```

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove branches

```
for(int i = 0; i < 10; i++) {
   int b = 1+2;
}
```

```
   set reg0 to 0
label L3
   compare reg0 to 9
   jump if greater to L2
   set reg1 to 3
   add 1 to reg0
   jump to L3
label L2
```

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove branches

```
for(int i = 0; i < 10; i++) {
   int b = 1+2;
}
```

```
   set reg0 to 0
label L3
   compare reg0 to 9
   jump if greater to L2
   set reg1 to 3
   add 1 to reg0
   jump to L3
label L2
```

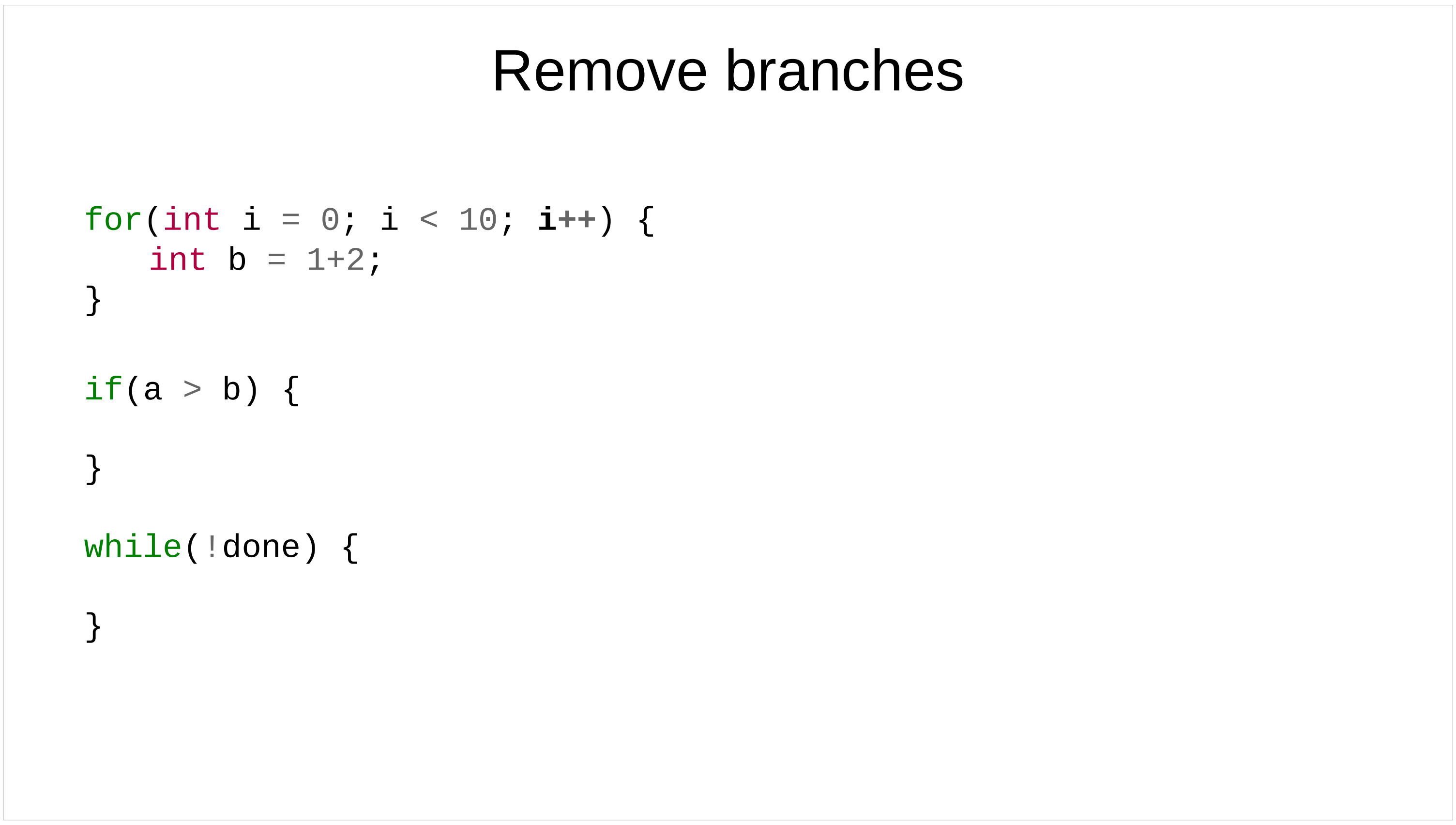
Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove branches

```
for(int i = 0; i < 10; i++) {
    int b = 1+2;
}
```

```
    set reg0 to 0
label L3
    compare reg0 to 9
    jump if greater to L2
    set reg1 to 3
    add 1 to reg0
    jump to L3
label L2
```

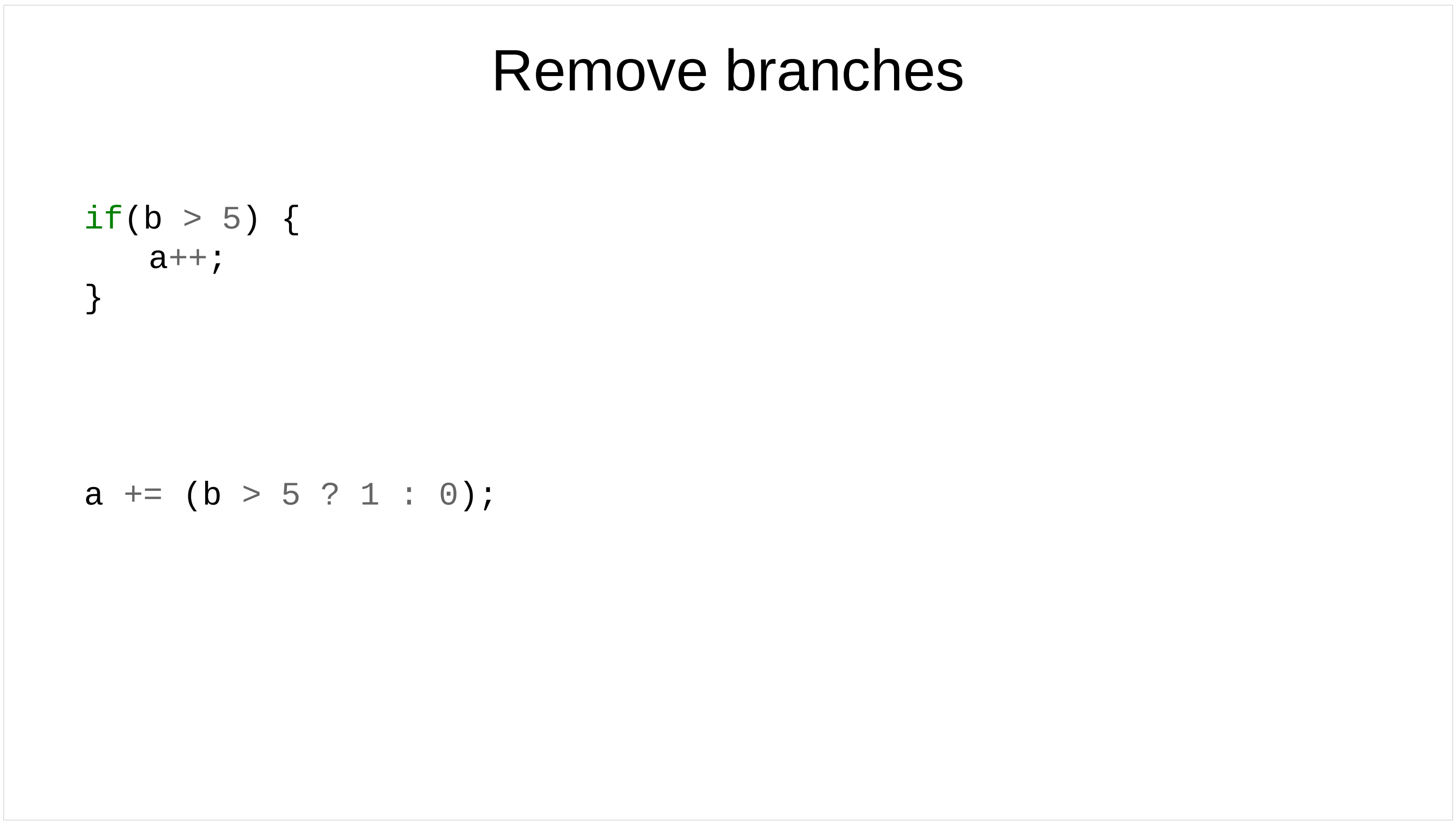
Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg



Remove branches

```
for(int i = 0; i < 10; i++) {
   int b = 1+2;
}

if(a > b) {

}

while(!done) {

}
```

Image from: https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg



Remove branches

```
if(b > 5) {
   a++;
}
```

```
a += (b > 5 ? 1 : 0);
```

Image from:
https://commons.wikimedia.org/wiki/File:Ditch_Witch_HT330_Track_Trencher_Stationary.jpg

Remove computation

Wrap expensive checks in cheaper ones

Image is screenshot from this video: https://www.youtube.com/watch?v=1wGcNzFd3as

```
for model in scene:
  if ray.intersects(model):
    doOuchie()
```

```
for model in scene:
  if ray.intersects(model.boundingBox):
    if ray.intersects(model):
      doOuchie()
```

Use integers instead of floats when you can

The compiler has an easier time optimising integer computations because there are no issues with precision loss

Organise switch statements by the most common case

```
switch() {
   case unrelated0:
      break;
   case unrelated1:
      break;
   case unrelated2:
      break;
   case unrelated3:
      break;
   case unrelated4:
      break;
   case mostCommon:
      break;
}
```

Organise switch statements by the most common case

```
switch() {
    case mostCommon:
        break;
    case unrelated0:
        break;
    case unrelated1:
        break;
    case unrelated2:
        break;
    case unrelated3:
        break;
    case unrelated4:
        break;
}
```

Move loops into functions if possible

```
void doSomethingExpensive() {

}

void doSomethingElse() {
    for(int i = 0; i < 1000000; i++) {
        doSomethingExpensive();
    }
}
```

Move loops into functions if possible

```
void doSomethingExpensive() {

}

void doSomethingElse() {
    for(int i = 0; i < 1000000; i++) {
        doSomethingExpensive();
    }
}
```

1) Push function’s stack frame
2) Evaluate arguments
3) Jump to function code
4) Execute function
5) Jump back
6) Pop function stack frame

Move loops into functions if possible

```
void doSomethingExpensive() {
    for(int i = 0; i < 1000000; i++) {

    }
}

void doSomethingElse() {
    doSomethingExpensive();
}
```

1) Push function’s stack frame
2) Evaluate arguments
3) Jump to function code
4) Execute function
5) Jump back
6) Pop function stack frame

Inline functions

(especially small ones)

Cache the loop condition variable

Remove computations from loop bodies

Use switch instead of if/else chains

Some higher level patterns

Exploit properties of your input data

Do actions less frequently

Batching

Try to a number of computations at once, rather than some at a time. Can save overhead.

Caching

Use memory to hold on to expensive values you computed previously.

Example: thread pool, length of a string

Specialisation

Opposite of generalisation.

Make things faster by making assumptions

Do computations in larger chunks

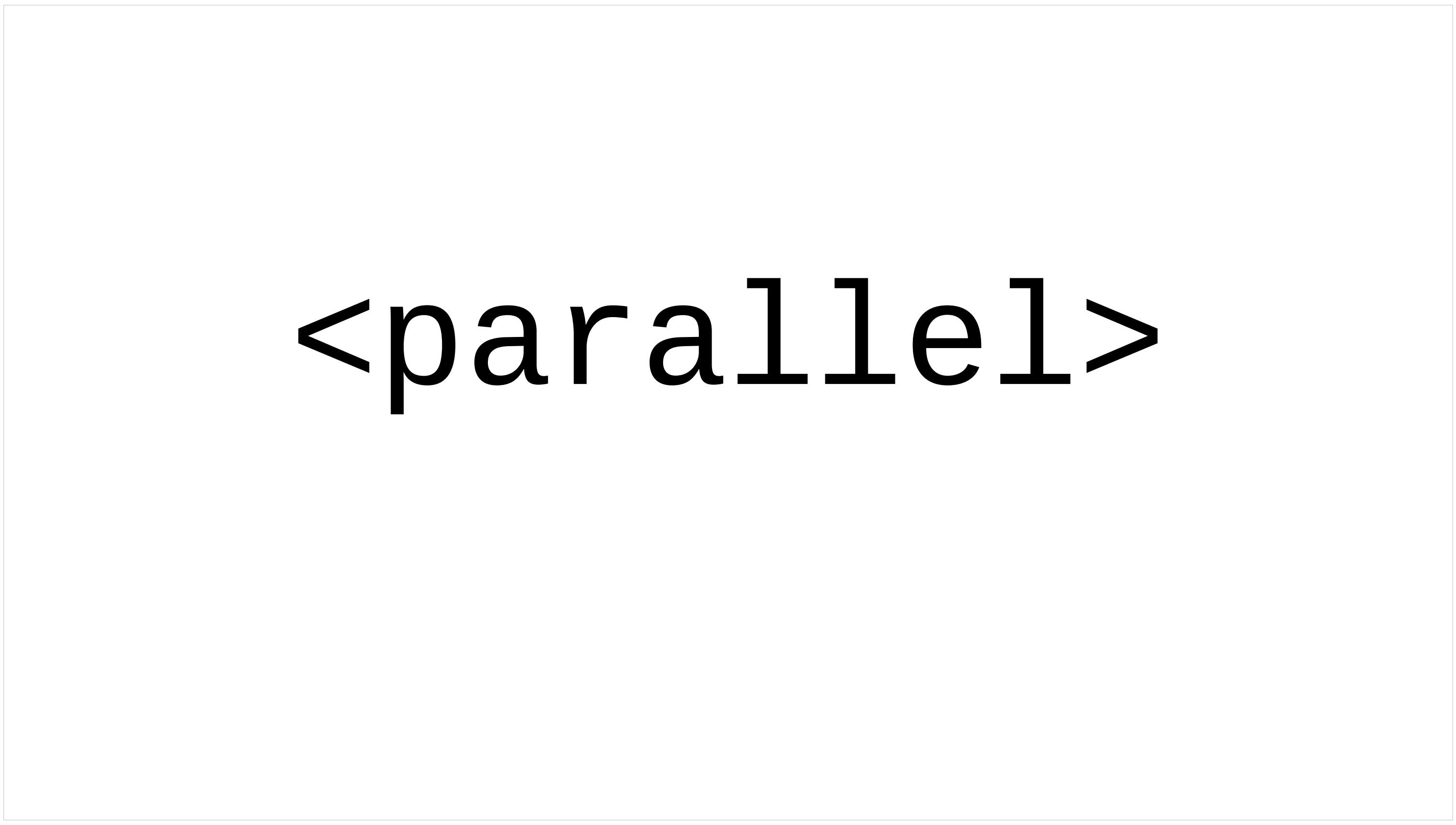
A recurring theme in optimisations across the board is that processing larger amounts of data gives you the benefits of economy of scale.

Generally not worth it:

Implementing your own std memory manager

Focussing on individual instructions

</serial>



<parallel>

This is cool. There’s so much stuff to talk about here!

Where do we even start?

What kinds of parallel systems do we have anyway?

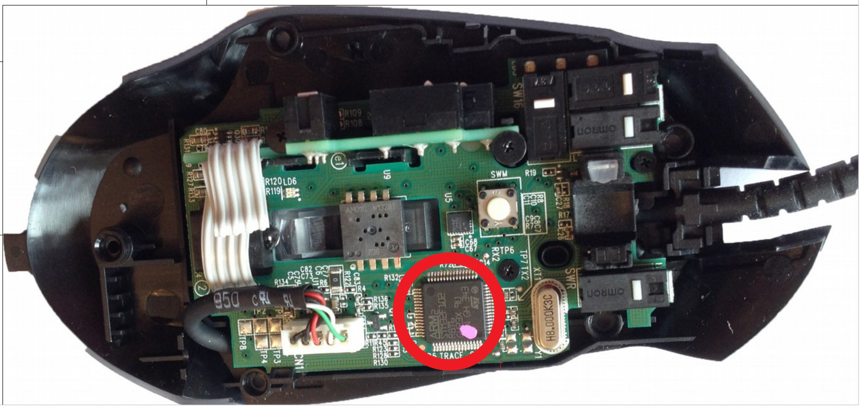
Classifying parallel systems:

Flynn’s Taxonomy

Proposed by Michael J. Flynn in 1966

	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	SISD	MISD	
Multiple Data Streams	SIMD (SIMT)	MIMD	SPMD

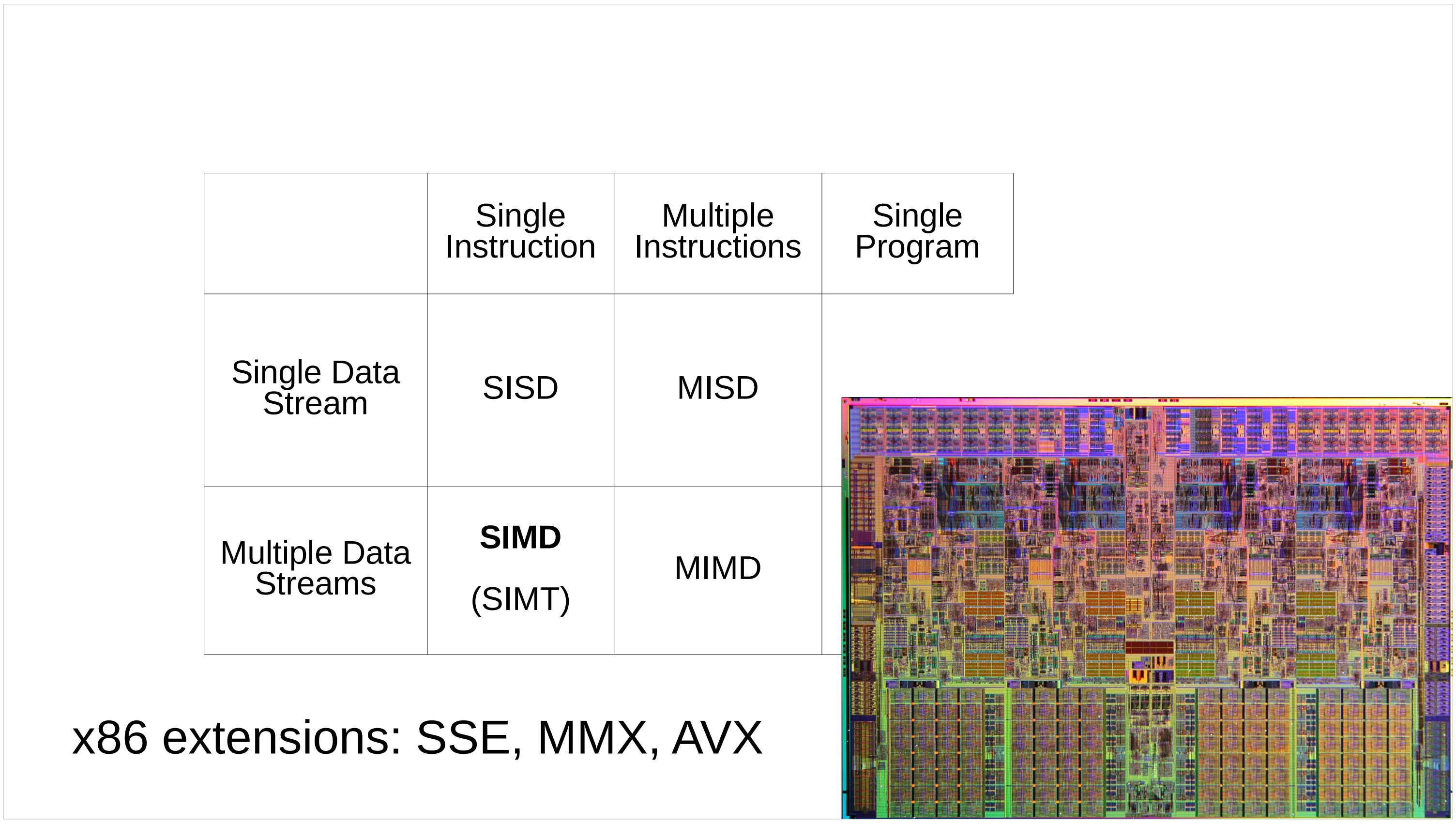
	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	**SISD**	MISD	
Multiple Data Streams	SIMD (SIMT)		



Single Instruction Single Data: your average “dumb” single-core processors. They process one instruction at a time, and each instruction is executed on one piece of data.

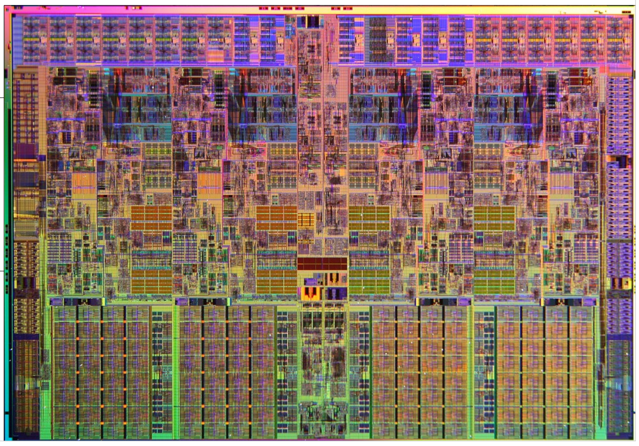
Good examples are microcontrollers, which tend to be single core machines. Even super low end CPUs are multicore nowadays.

Image from: https://www.overclock.net/forum/375-mice/1504917-logitech-g402-hyperion-fury-gaming-mouse-review-ino.html



	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	SISD	MISD	
Multiple Data Streams	**SIMD** (SIMT)	MIMD	

x86 extensions: SSE, MMX, AVX

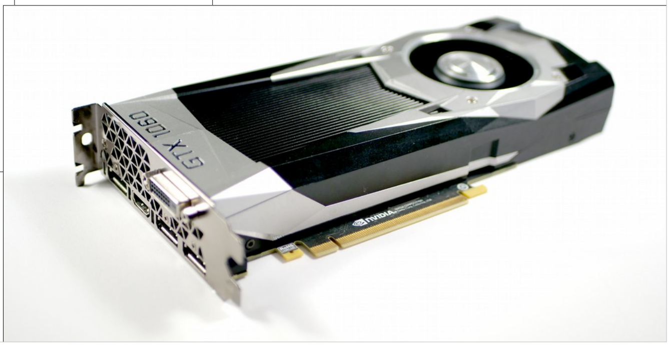


SIMD instructions are classically called “vector processors”. The idea is that arithmetic operations tend to be executed many times on different pieces of data. Rather than create separate CPU cores, you instead share control logic such as the instruction decoder, and execute the instruction on multiple pieces of data simultaneously.

Nowadays, you will find this “style” of instruction integrated into modern processors. On x86, you will find these instructions in the SSE, MMX, and AVX extensions.

SIMD processors execute their instructions in lockstep. This means that each instruction performs multiple additions/multiplications/whatnot, they MUST all be executed at once (or wait until all can be executed).

	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	SISD	MISD	
Multiple Data Streams	SIMD **(SIMT)**	MIMD	

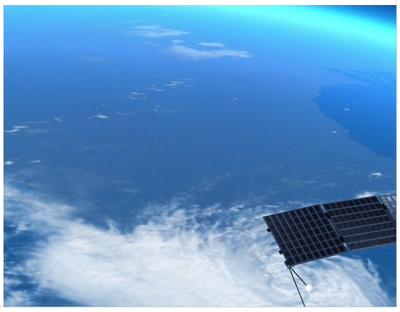


A variation of SIMD machines is the Single Instruction Multiple Thread category. This was not present in the taxonomy Flynn proposed, but has implicitly been added

Note that SIMD and SIMT are nearly equivalent. The only difference is that in SIMT, each processed value belongs to a separate thread.

Image credit: https://www.eurogamer.net/articles/digitalfoundry-2018-09-05-best-graphics-cards-2018-7001

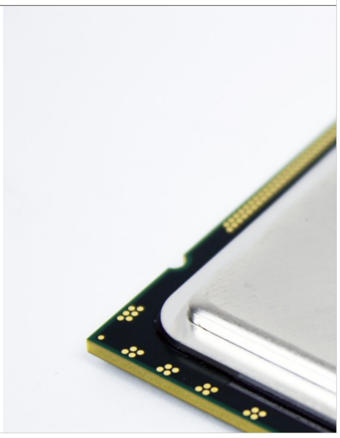
	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	SISD	**MISD**	
Multiple Data Streams	SIMD (SIMT)	MIMD	SPMD



MISD is a very rare processor type. One of the main places you will find it "in the wild" is in systems that HAVE to be reliable, such as satellites. It's pretty hard to fly out in space to service one, so once it's up there, it HAS to keep working. They often have multiple processors executing the same instructions, where the results are compared to make sure no fault has occurred, which can happen due to factors such as interstellar radiation.

Image credit: https://africa.cgtn.com/2017/01/04/ethiopia-to-launch-own-satellite/

	Single Instruction	Multiple Instructions	Single Program
Single Data Stream	SISD	MISD	
Multiple Data Streams	SIMD (SIMT)	**MIMD**	SPMD



CPUs have multiple cores nowadays. Each core can execute instructions independently, and cores can work on different pieces of data

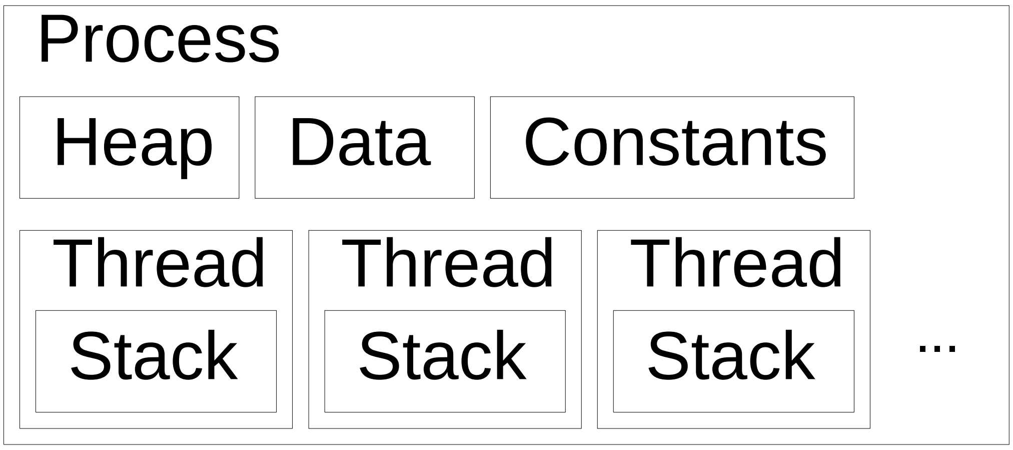
Image credit: https://create.pro/blog/cores-faster-cpu-clock-speed-explained/

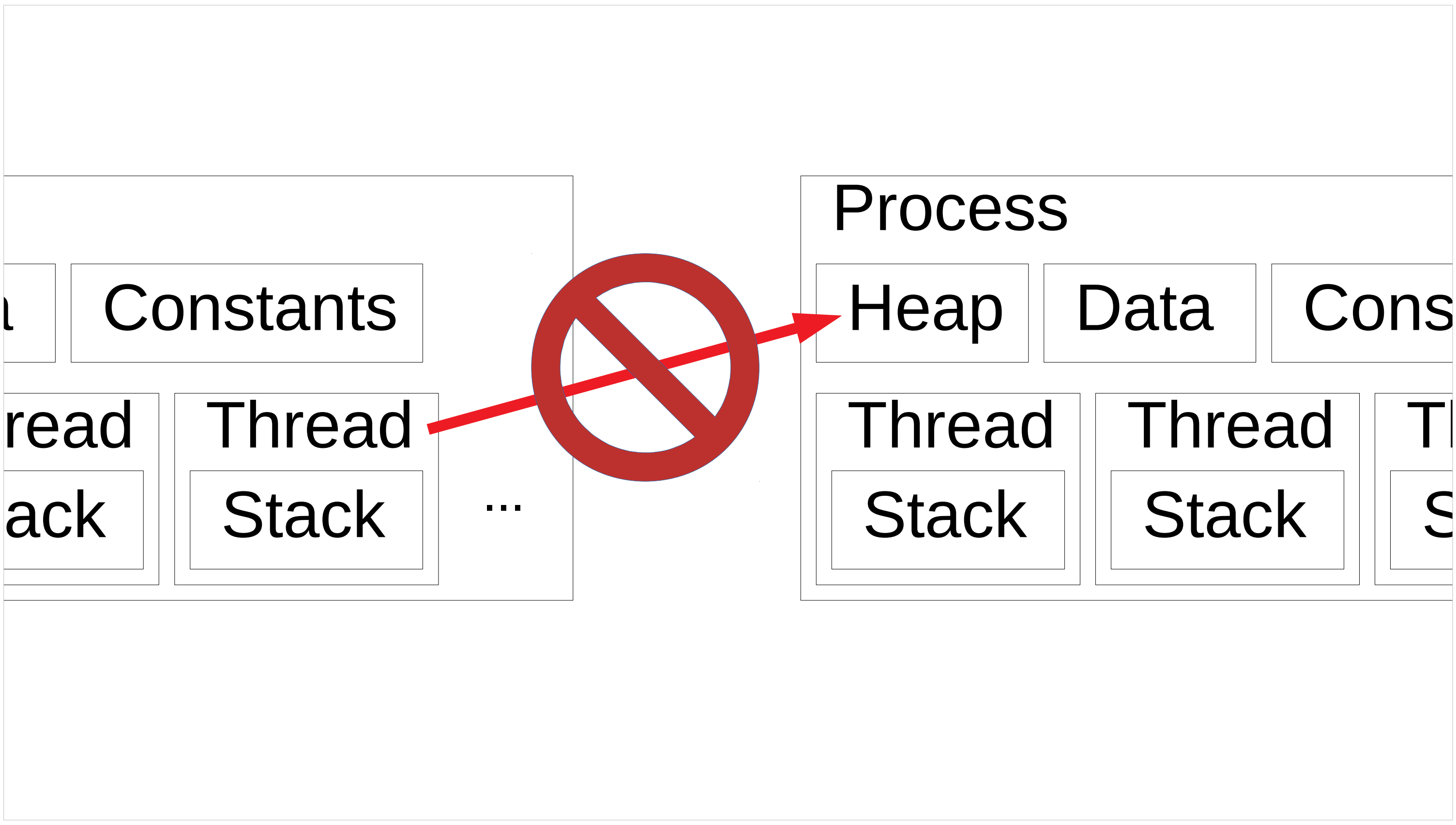
Multiple Data Streams	SIMD (SIMT)	MIMD	**SPMD**

MPI

Single Program Multiple Data is essentially MPI; you run the same program on multiple physical machines, usually in the context of a supercomputer. These individual instances in turn execute the Same Program on Different pieces of Data.

Thread versus Process


Process
Heap
Data
Constants
Thread
Stack
Thread
Stack
Thread
Stack
...

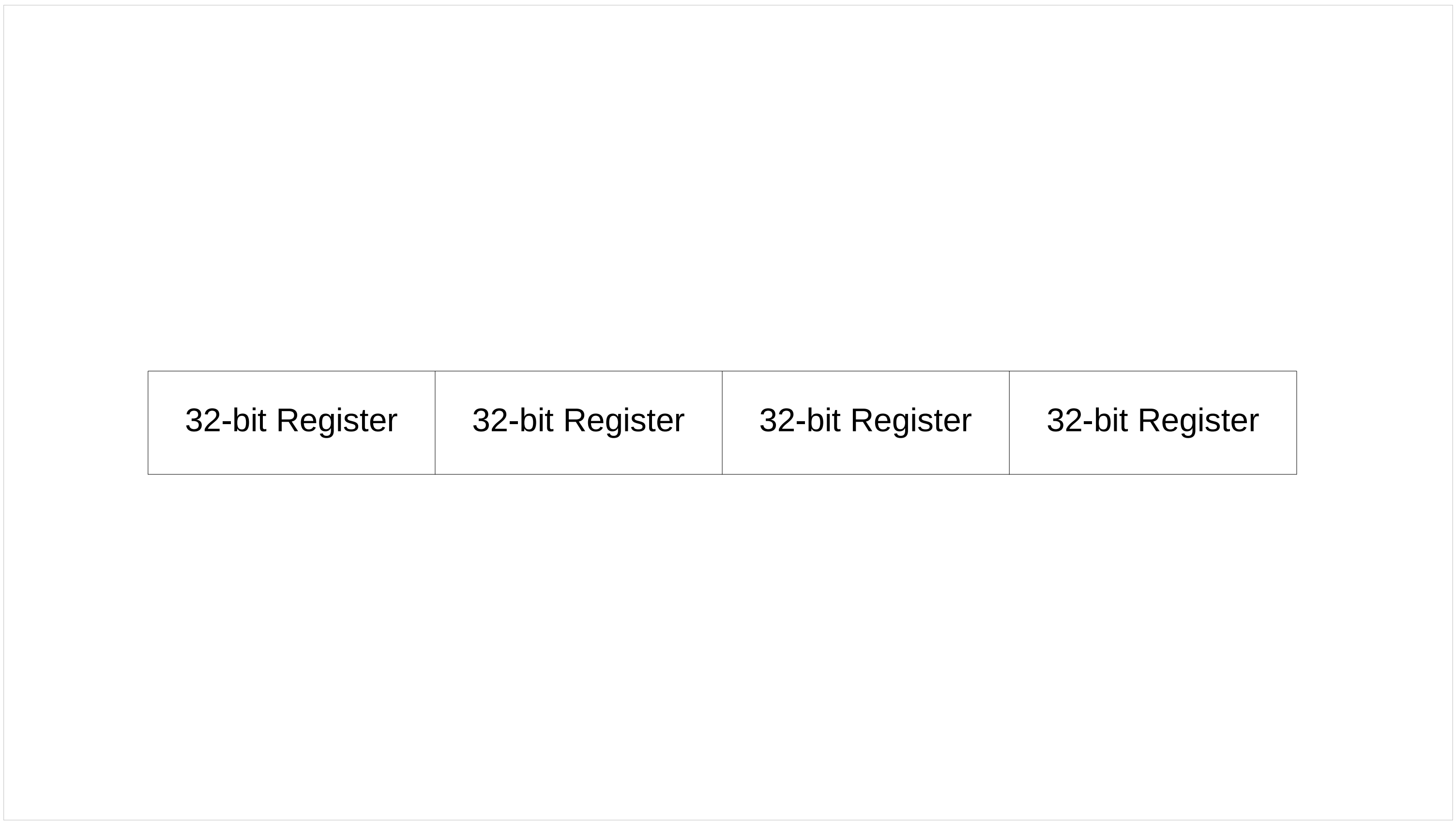


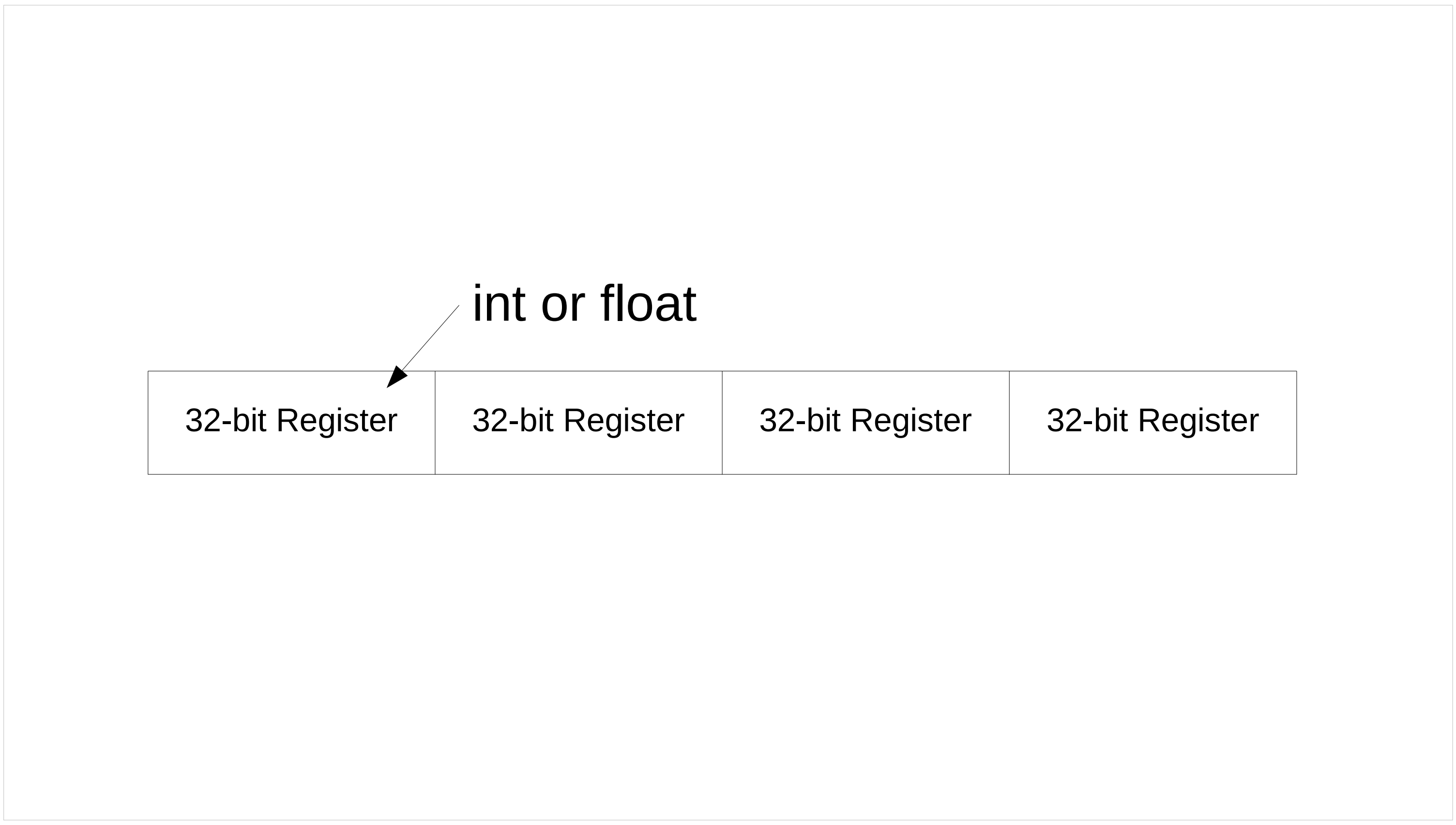
Unrelated note: the operating system also takes into account which processing core a thread was assigned to, because the cache still has the thread's variables inside it.

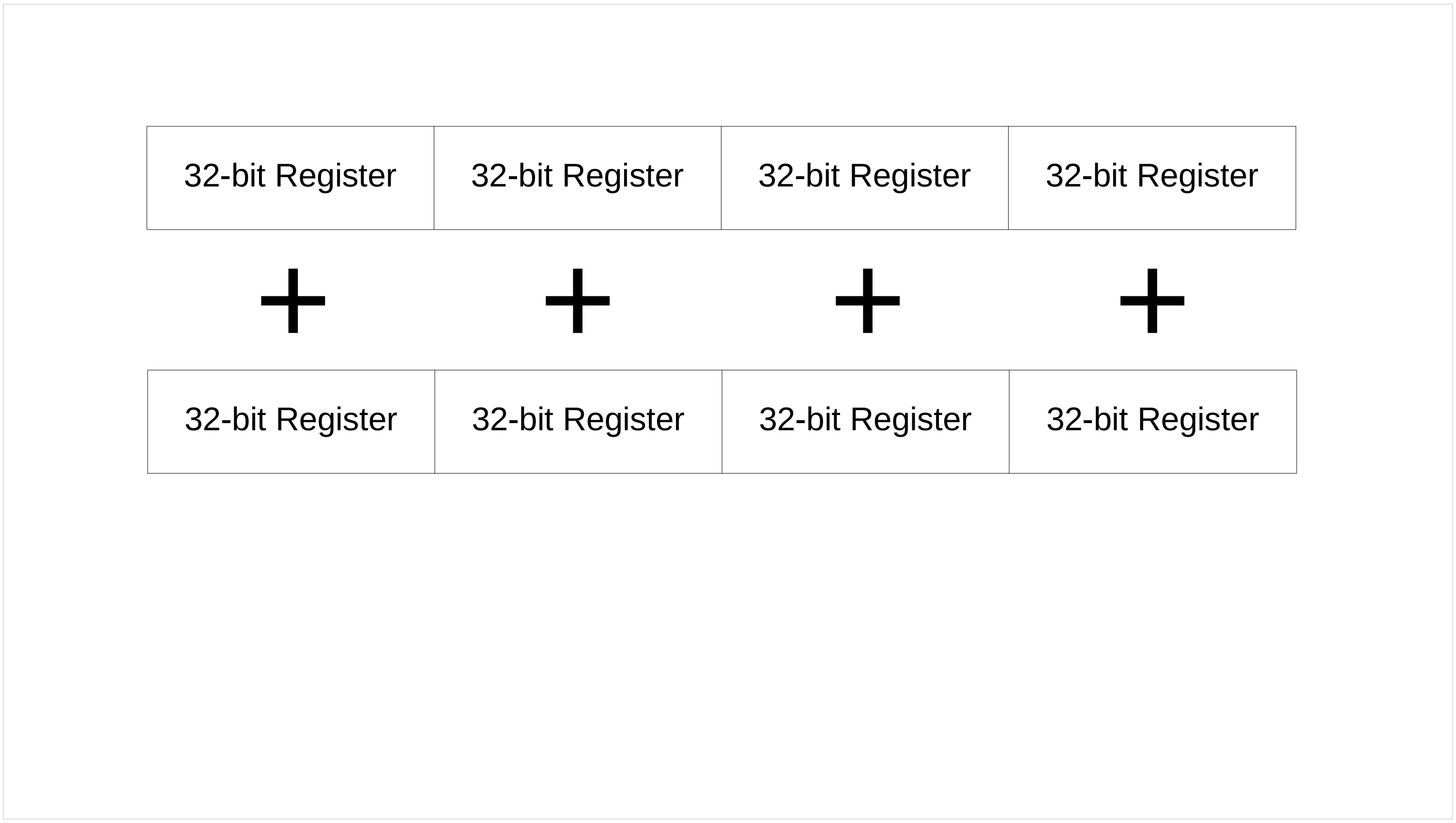
.. You mentioned x86 processors have SIMD instructions.

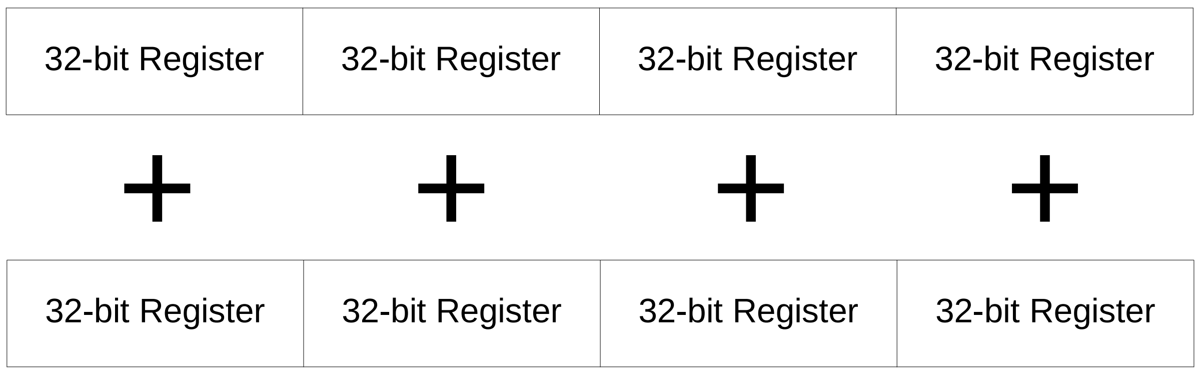
Streaming SIMD Extensions (SSE)

Initially introduced on the pentium III


32-bit Register
32-bit Register
32-bit Register
32-bit Register


int or float
32-bit Register
32-bit Register
32-bit Register
32-bit Register


32-bit Register
32-bit Register
32-bit Register
32-bit Register
+
+
+
+
32-bit Register
32-bit Register
32-bit Register
32-bit Register



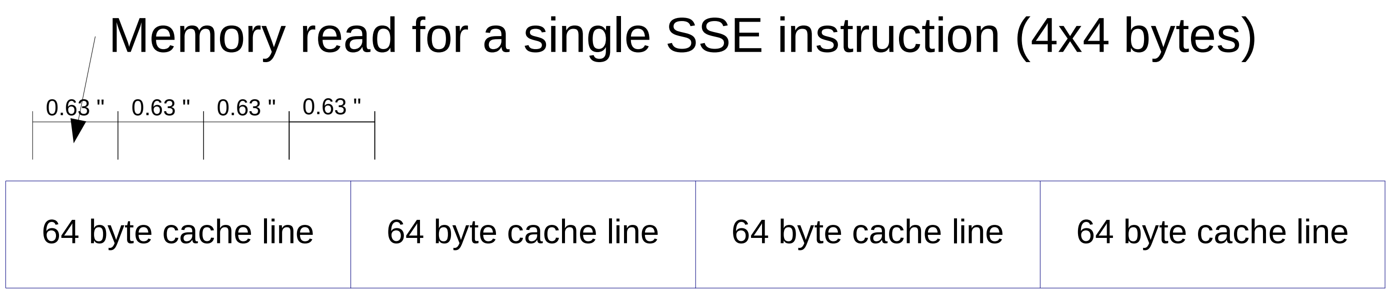
Throughput	Normal	Vector
Addition	`add` (4 / cycle)	`addps` (2 / cycle)
Multiplication	`mul` (1 / cycle)	`mulps` (2 / cycle)
Division	`div` (0.17 / cycle)	`divps` (0.33 / cycle)

Works great on lots of chunky computations

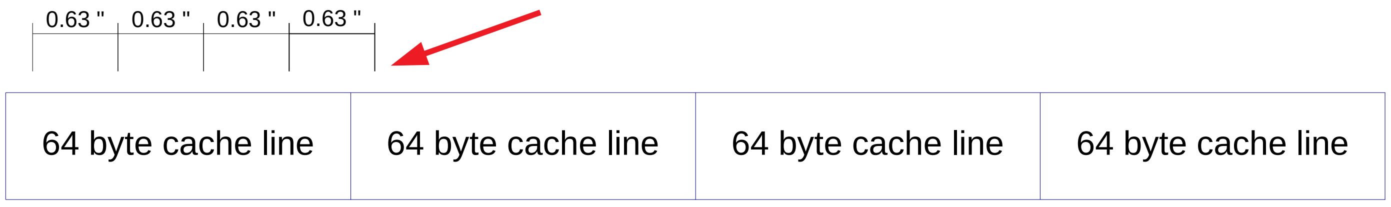
Downsides and caveats:

- Requires aligned memory
- All or nothing instructions
- Best to read and write once

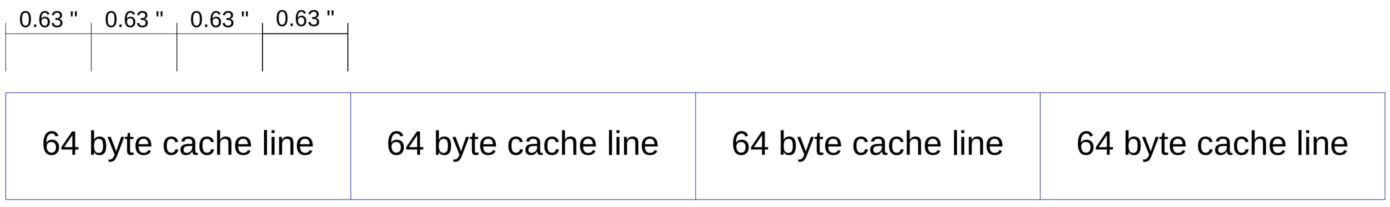
Aligned Memory



Aligned Memory



Aligned Memory



```
union sse_float4 {
    float __attribute__ ((vector_size (16))) vector;
    float elements[4];
};
```

This is non-standard C++, specific to GCC!

All or nothing instructions:

No branches

No option for conditional operations

Best to read and write once:

Minimises bandwidth

Import

```
#include <iostream>
#include <chrono>

union sse_float4 {
    float __attribute__ ((vector_size (16))) vector;
    float elements[4];
};

int main(int argc, char** argv) {

   const long iterationCount = 100000000;

   sse_float4* values = new sse_float4[iterationCount];
   float* regularValues = new float[4l * iterationCount];

   for(long i = 0; i < iterationCount; i++) {
      values[i].elements[0] = i;
      values[i].elements[1] = i + 1;
      values[i].elements[2] = i - 1;
      values[i].elements[3] = i + 4;

      regularValues[i] = i;
   }
```

```cpp
auto start = std::chrono::high_resolution_clock::now();
for(long i = 0; i < (4l * iterationCount) - 4; i++) {
   regularValues[i] =
      regularValues[i] * regularValues[i + 4];
}
auto end = std::chrono::high_resolution_clock::now();

auto timeTaken =
   std::chrono::duration_cast<std::chrono::milliseconds>
   (end – start).count();

std::cout << "Basic time: " << timeTaken << std::endl;
```

```
    start = std::chrono::high_resolution_clock::now();
    for(long i = 0; i < iterationCount - 1; i++) {
       sse_float4 current = values[i];
       sse_float4 next = values[i + 1];
       values[i].vector = current.vector * next.vector;
    }
    end = std::chrono::high_resolution_clock::now();

    timeTaken =
       std::chrono::duration_cast<std::chrono::milliseconds>
       (end – start).count();

    std::cout << "SSE time: " << timeTaken << std::endl;

 }
```

```
g++ sse.cpp -o sse -march=native -O3
```

Basic time: 510 ms

SSE time: 105 ms

.. Technically single-threaded

How to verify whether you’re using SSE/AVX instructions:

```
g++ sse.cpp -o sse -march=native -O3 -S
```

Look for:

`vaddps`, `vsubps`, `vmulps`, `vdivps`

Next week: Supercomputers!