Introduction to

SIMD AND (AUTO)VECTORIZATION

Felix Kutzner (QPR Technologies)
C++ User Group Karlsruhe

Agenda

Agenda

Part I: Introduction to SIMD Programming

Agenda

Part I: Introduction to SIMD Programming

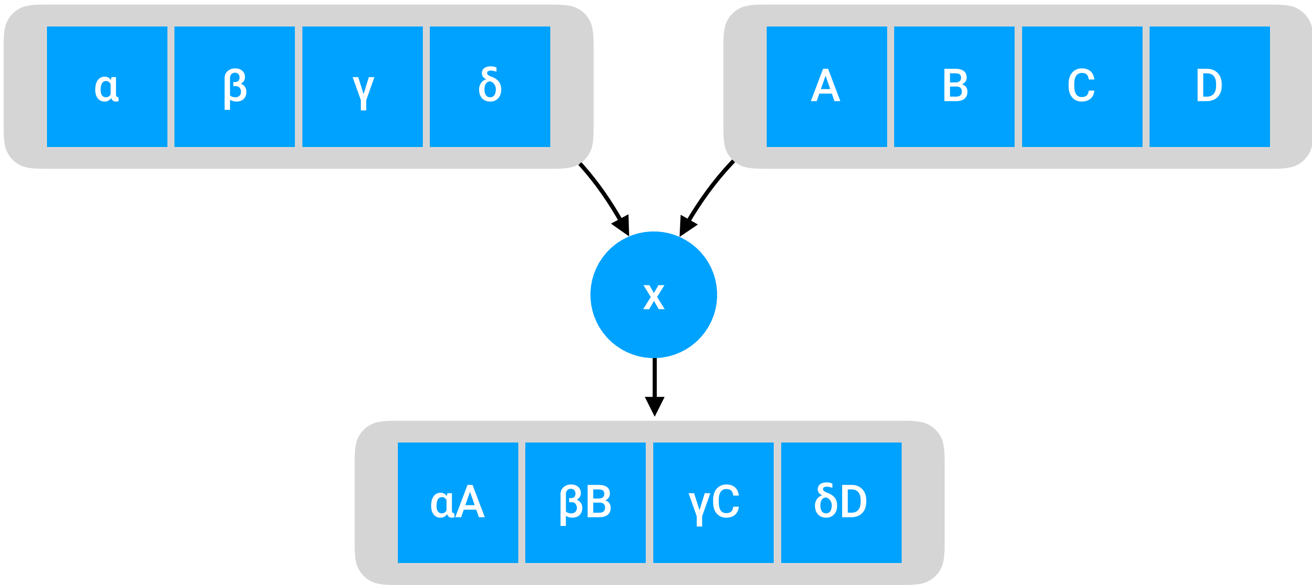
Part II: Introduction to Vectorization

INTRODUCTION TO SIMD PROGRAMMING

Flynn’s Taxonomy

	Single Instruction	**Multiple Instruction**
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

SIMD



Popular Implementations

Popular Implementations

- x86_64, x86: AVX, SSE, MMX

Popular Implementations

- x86_64, x86: AVX, SSE, MMX
- ARM: NEON

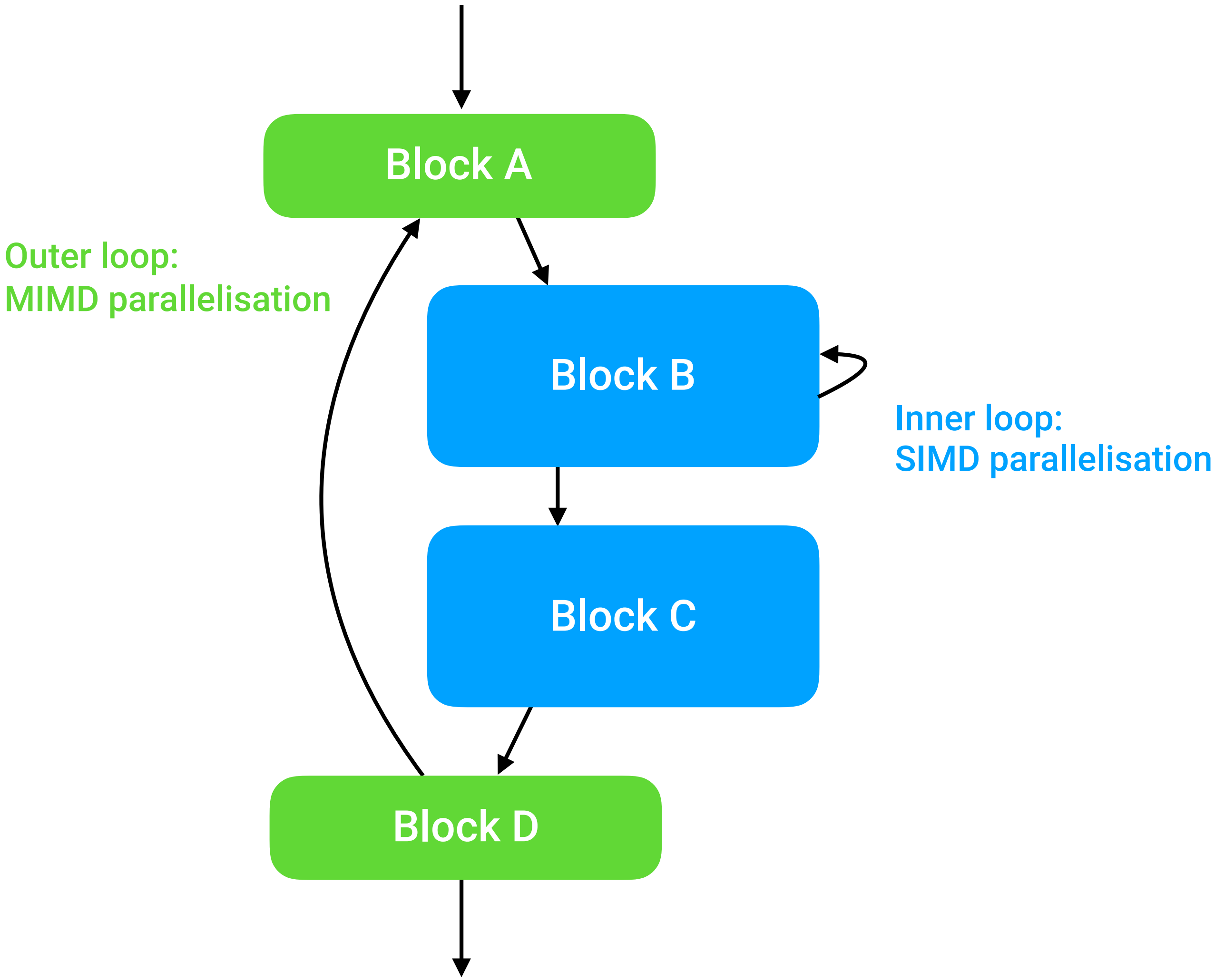
Popular Implementations

- x86_64, x86: AVX, SSE, MMX
- ARM: NEON
- PowerPC: AltiVec

Popular Implementations

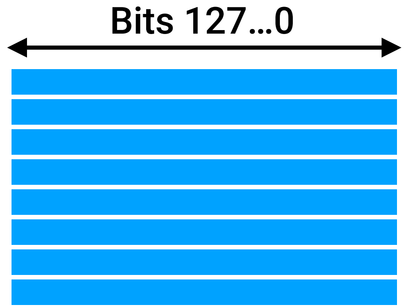
- x86_64, x86: AVX, SSE, MMX
- ARM: NEON
- PowerPC: AltiVec
- GPGPU

Using Small-Vector SIMD

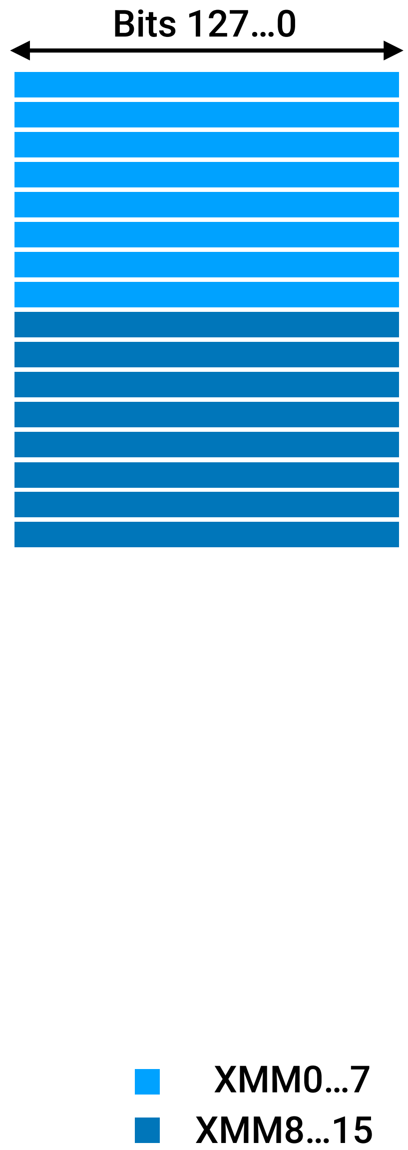


SSE/AVX Registers

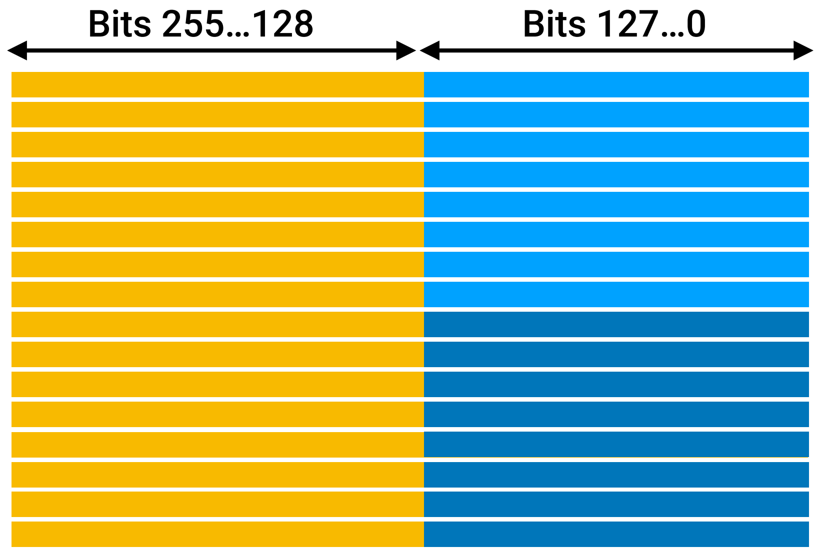
SSE/AVX Registers



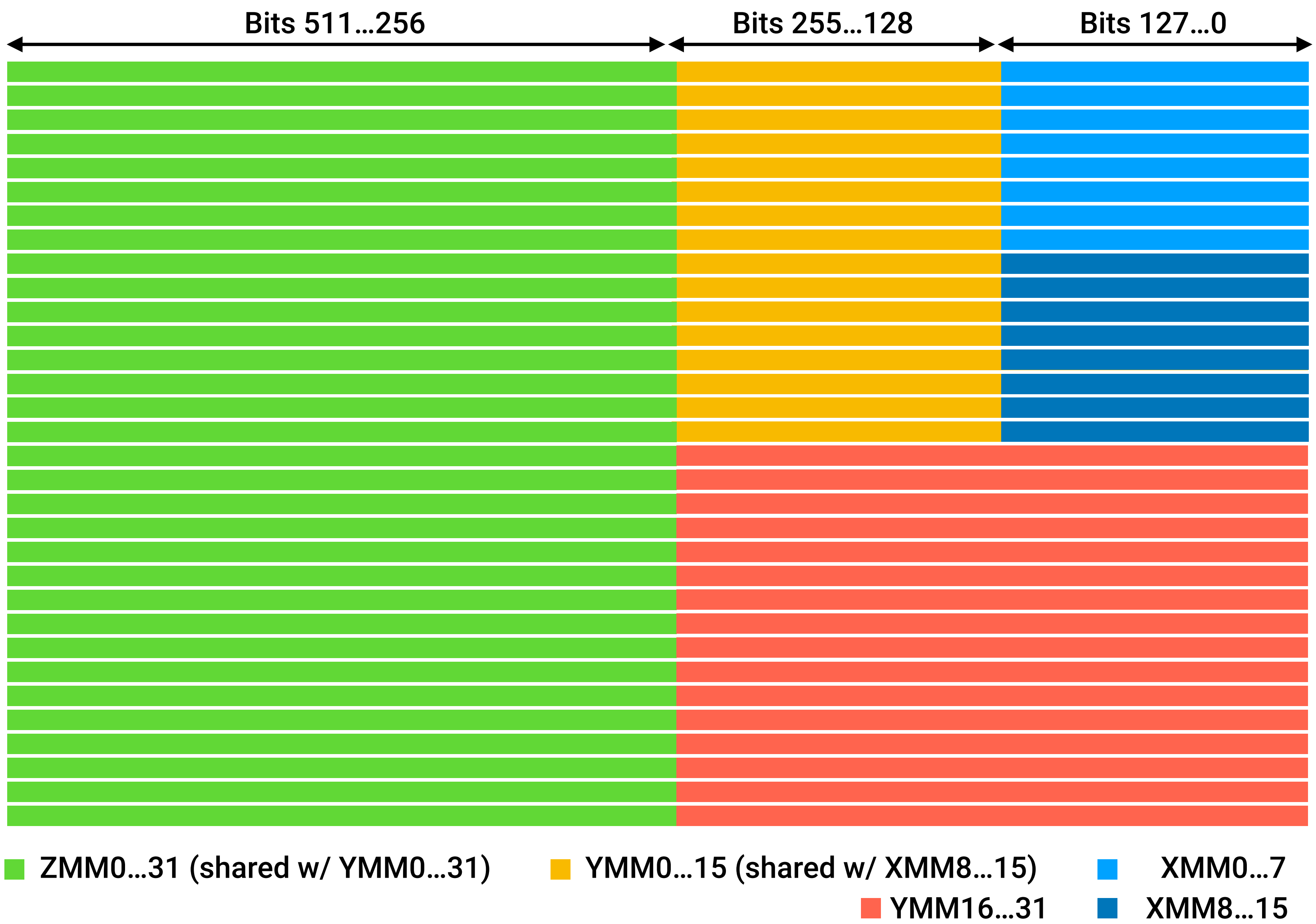
SSE/AVX Registers



SSE/AVX Registers

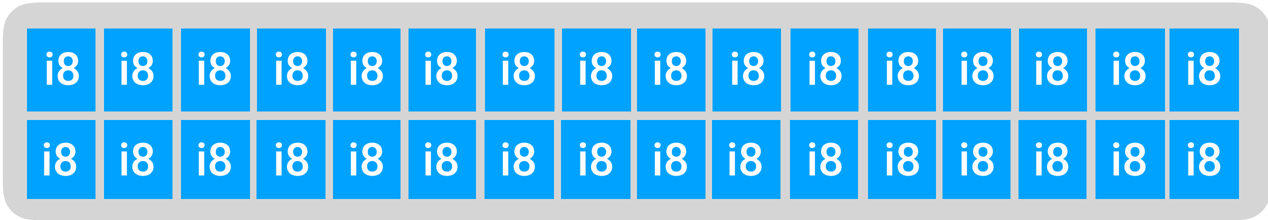


SSE/AVX Registers



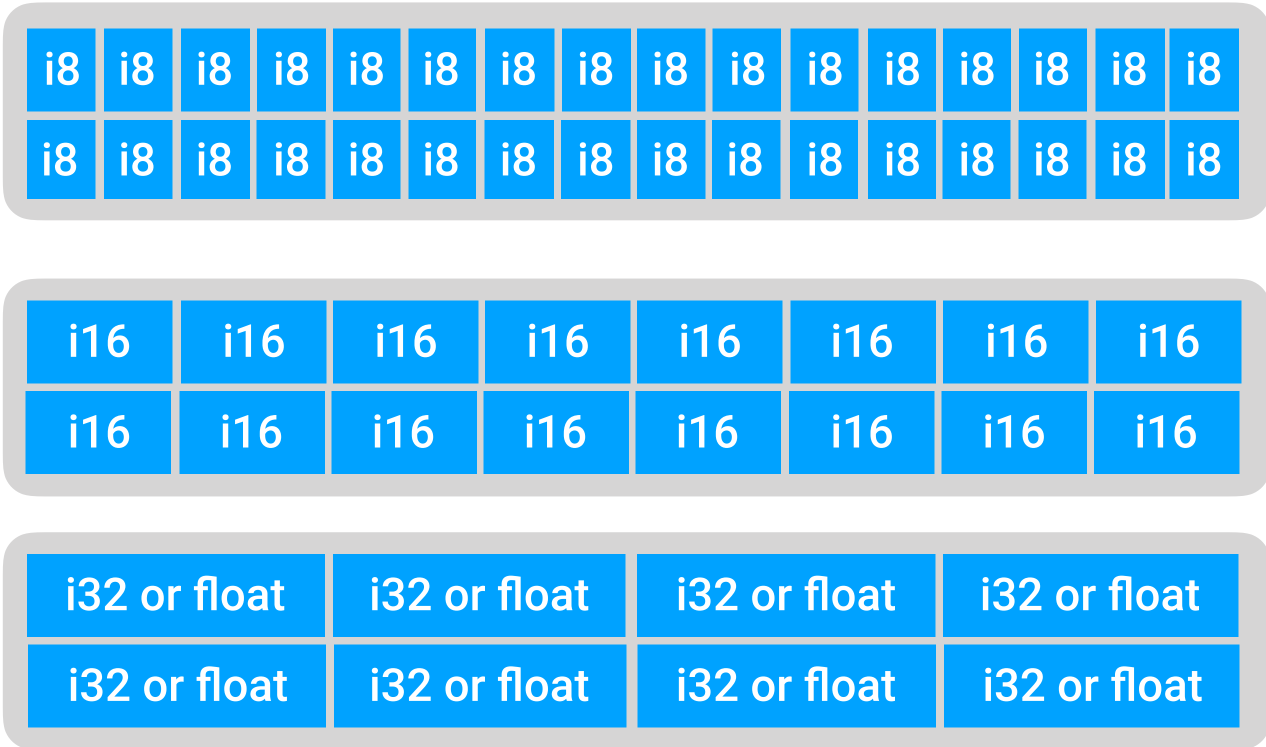
AVX Register Packing

AVX Register Packing

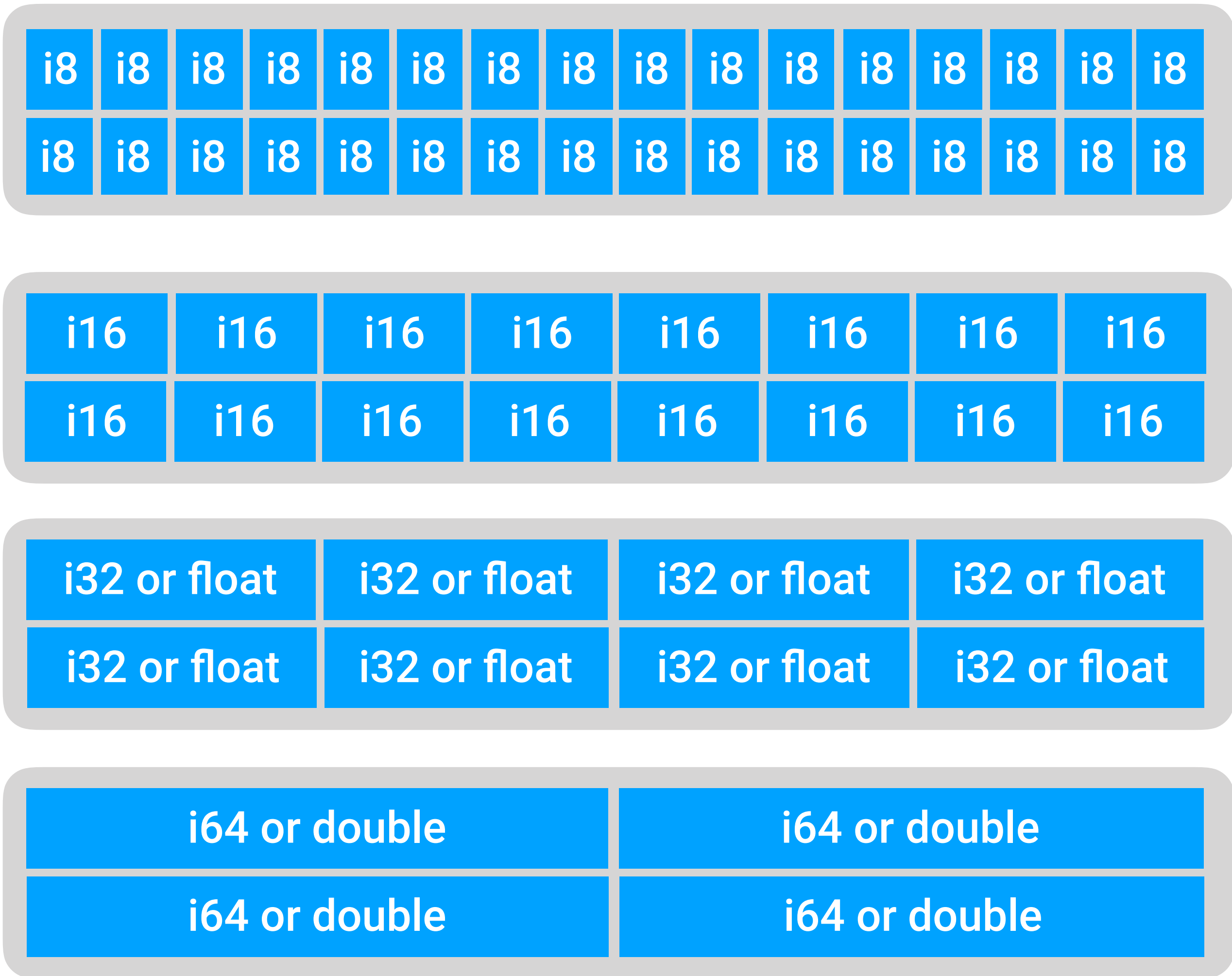


AVX Register Packing

AVX Register Packing



AVX Register Packing



Instructions

Instructions

- Move, scatter & gather
- Arithmetic
- Bitwise AND, OR, ...
- Conditionals
- Shuffling & permutation
- Insertion & extraction
- Reduction
- …

Move

Move

`vmovaps`

Move

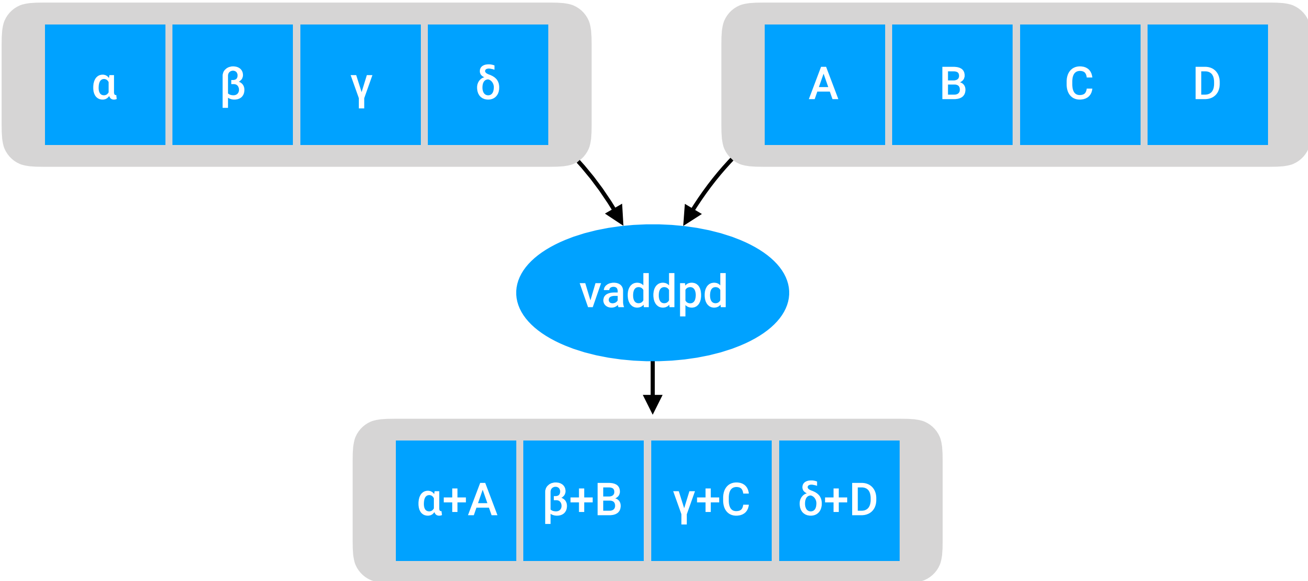
`vmovaps`

`vmovupd`

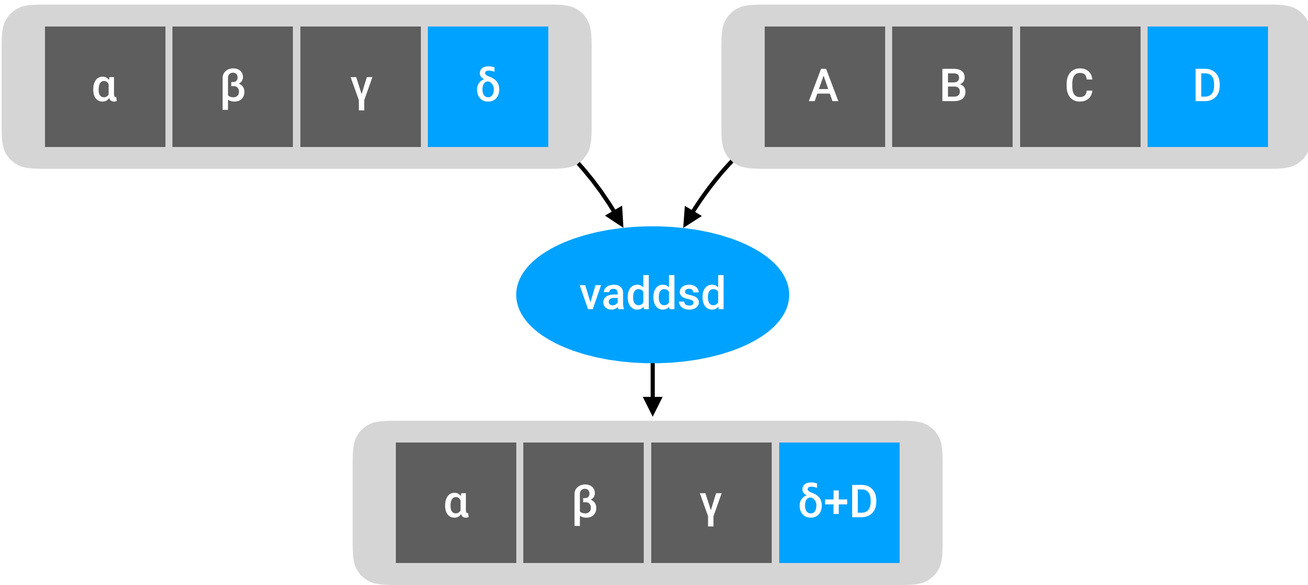
Move

```
vmovaps

vmovupd

vmovsd
```

Arithmetic



Arithmetic



Conditionals

Conditionals

Given

A =	a1	a2	a3	a4
B =	b1	b2	b3	b4

Compute

X =	min (a1,b1)	min (a2,b2)	min (a3,b3)	min (a4,b4)

Conditionals

Given

A =	a1	a2	a3	a4
B =	b1	b2	b3	b4

Compute

X =	min (a1,b1)	min (a2,b2)	min (a3,b3)	min (a4,b4)

Solution: Compute mask

M =	M1	M2	M3	M4

with M*i* = 0xFFF…FFF if *ai* ≤ b*i*,
M*i* = 0x000…000 otherwise

Conditionals

Given

A = | a1 | a2 | a3 | a4 |

B = | b1 | b2 | b3 | b4 |

Compute

X = | min (a1,b1) | min (a2,b2) | min (a3,b3) | min (a4,b4) |

Solution: Compute mask

M = | M1 | M2 | M3 | M4 |

with M*i* = 0xFFF...FFF if *ai* ≤ b*i*,
M*i* = 0x000...000 otherwise

X = (A AND M) OR (B AND ~M)

Conditionals

Conditionals

`vcmppd`

Conditionals

vcmppd

vcmpsd

Conditionals

`vcmppd`

`vcmpsd`

`vcmpw`

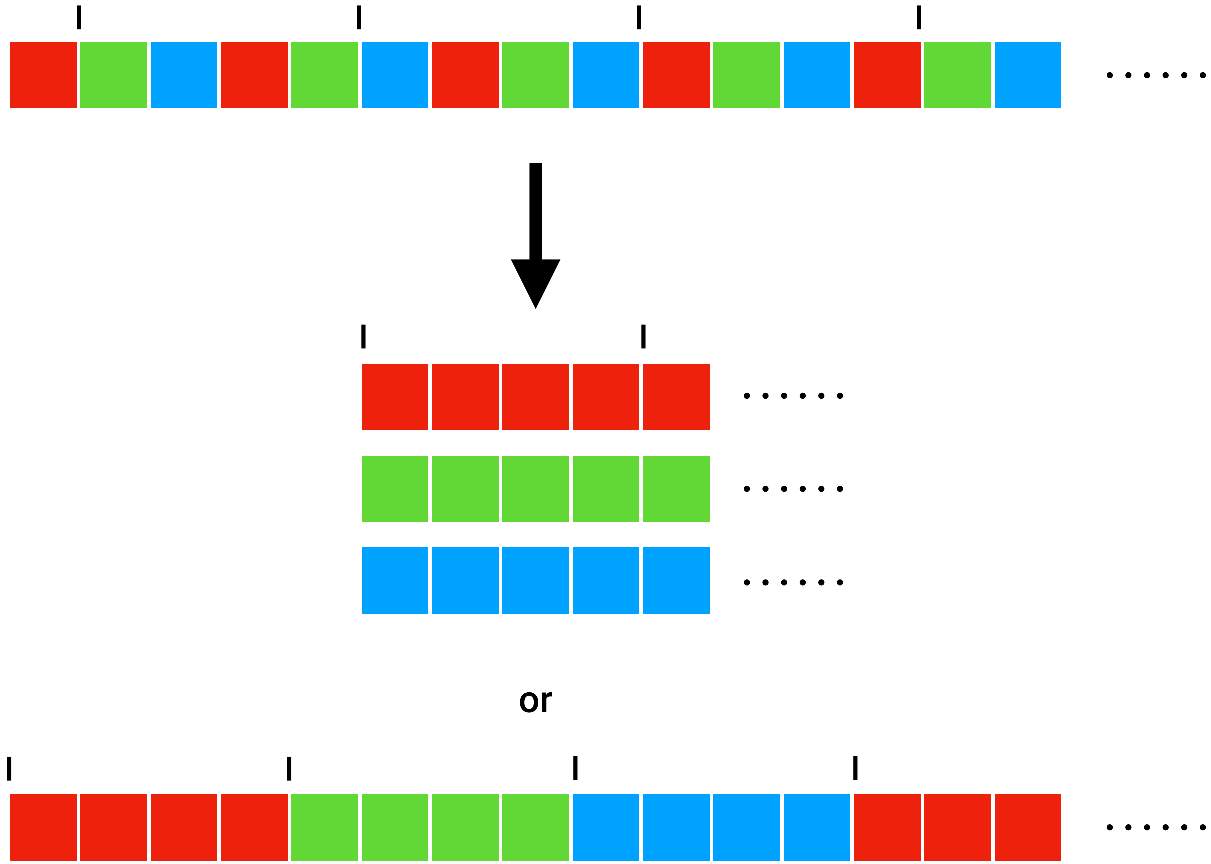
AVX can be hard to read

`vpunpcklqdq`

Compiler Explorer is tremendously useful for this

godbolt.org

Data Layout



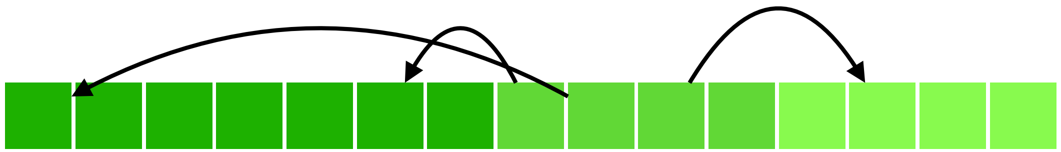
Alignment in C++

```
template<typename T, size_t VecSize>
struct alignas(VecSize) pack {
    static constexpr size_t size = VecSize/sizeof(T);
    static constexpr size_t vec_size = VecSize;

    std::array<T, size> x;
};
```

Data Dependencies

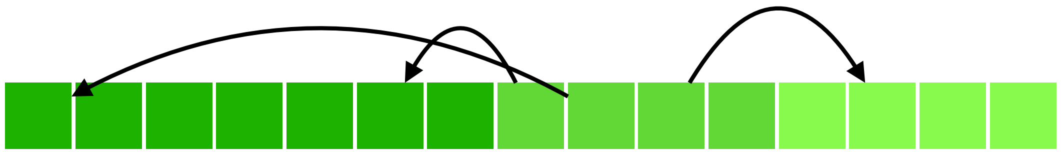
Data Dependencies

Hard to optimize:

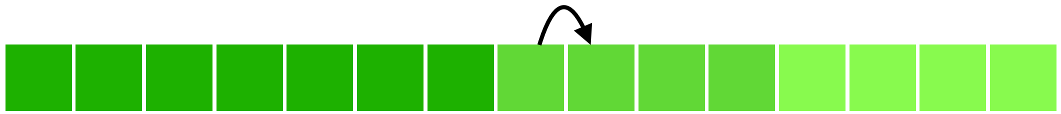


Data Dependencies

Hard to optimize:

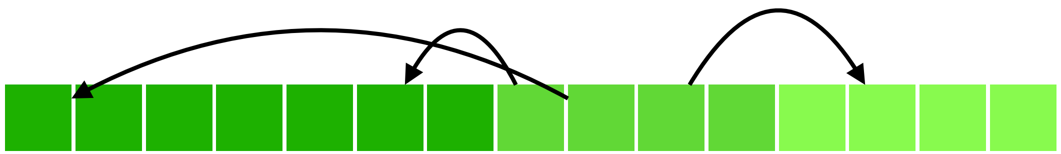


Sometimes problematic:

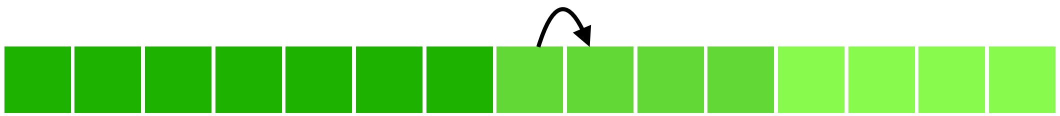


Data Dependencies

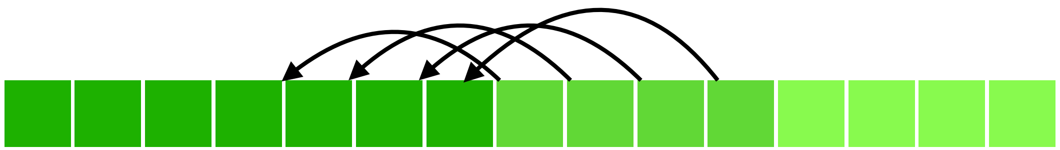
Hard to optimize:



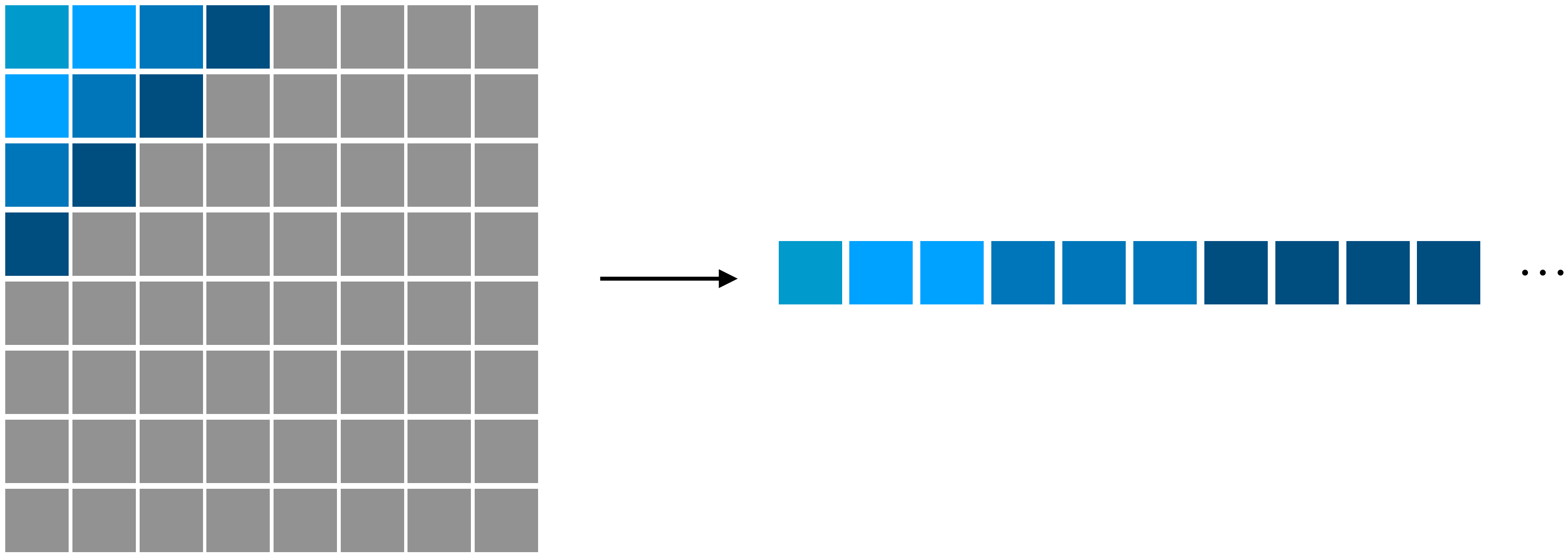
Sometimes problematic:



Convenient:



Data Dependencies



Options

Options

- (Write assembly code)

Options

- (Write assembly code)
- OpenCL / CUDA

Options

- (Write assembly code)
- OpenCL / CUDA
- Use compiler intrinsics

Options

- (Write assembly code)
- OpenCL / CUDA
- Use compiler intrinsics
- Use a library

Options

- (Write assembly code)
- OpenCL / CUDA
- Use compiler intrinsics
- Use a library
- Use OpenMP / rely on autovectorization

AVX/SSE Intrinsics Data Types

`__m<Len>[Ty]` e.g. `__m128i`

Vector length

64
128
256
512

Vector type

i: packed integers
d: packed double
(When omitted: packed float)

Intrinsics by Example

Pros & Cons: Intrinsics

Pros & Cons: Intrinsics

Benefits:

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions
- It’s not assembly

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions
- It's not assembly

Drawbacks:

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions
- It's not assembly

Drawbacks:

- Tied to SIMD implementation & version

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions
- It’s not assembly

Drawbacks:

- Tied to SIMD implementation & version
- Cumbersome to write and maintain

Pros & Cons: Intrinsics

Benefits:

- Almost full control over the generated machine code
- Can use obscure AVX instructions
- It’s not assembly

Drawbacks:

- Tied to SIMD implementation & version
- Cumbersome to write and maintain
- Need knowledge of target CPU microarchitecture

OpenMP by Example

Helpful Resources

Intel Intrinsics Guide
https://software.intel.com/sites/landingpage/IntrinsicsGuide

Agner Fog: Instruction Tables
www.agner.org/optimize/instruction_tables.pdf

Agner Fog: The microarchitecture of Intel, AMD and VIA CPUs
www.agner.org/optimize/microarchitecture.pdf

Intel® 64 and IA-32 Architectures Software Developer Manuals
https://software.intel.com/en-us/articles/intel-sdm

Chris Lomont: Introduction to Assembly Programming
https://software.intel.com/en-us/articles/introduction-to-x64-assembly

Introduction to SSE; SSE wrapper library *cvalarray*
http://sci.tuomastonteri.fi/programming/sse

Tools

Simple disassembler: objdump --disassemble

Disassembler & decompiler: Hopper Disassembler

Compiler Explorer: godbolt.org

Cache efficiency: cachegrind (included with valgrind)

Expensive tools: Intel VTune, IDA Pro

(AUTO)VECTORIZATION

Autovectorization

Autovectorization

```
#include <array>

class vec {
public:
  std::array<float, 4> m_pts;

  vec operator+(const vec& rhs) {
    vec result;
    for (int i = 0; i < 4; ++i) {
      result.m_pts[i] =
           m_pts[i] + rhs.m_pts[i];
    }
    return result;
  }
};
```

Autovectorization

```cpp
#include <array>

class vec {
public:
  std::array<float, 4> m_pts;

  vec operator+(const vec& rhs) {
    vec result;
    for (int i = 0; i < 4; ++i) {
      result.m_pts[i] =
           m_pts[i] + rhs.m_pts[i];
    }
    return result;
  }
};
```

clang 5, -std=c++14 -msse-4.1 -O3

```asm
vec::operator+(vec const&):

movups xmm1, xmmword ptr [rdi]
movups xmm0, xmmword ptr [rsi]
addps xmm0, xmm1
movaps xmmword ptr [rsp - 24], xmm0
movsd xmm1, qword ptr [rsp - 16]
ret
```

Vectorization Reports

Vectorization Reports

e.g. clang:

```
-Rpass="loop|vect" -Rpass-missed="loop|vect" -Rpass-analysis="loop|vect"
```

Vectorization Reports

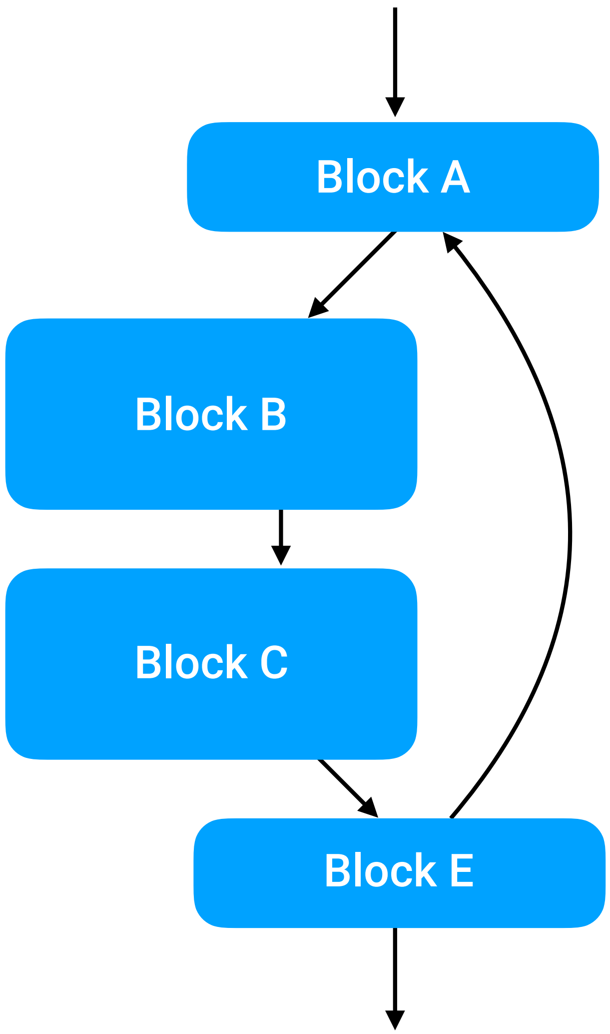
e.g. clang:

```
-Rpass="loop|vect" -Rpass-missed="loop|vect" -Rpass-analysis="loop|vect"
```

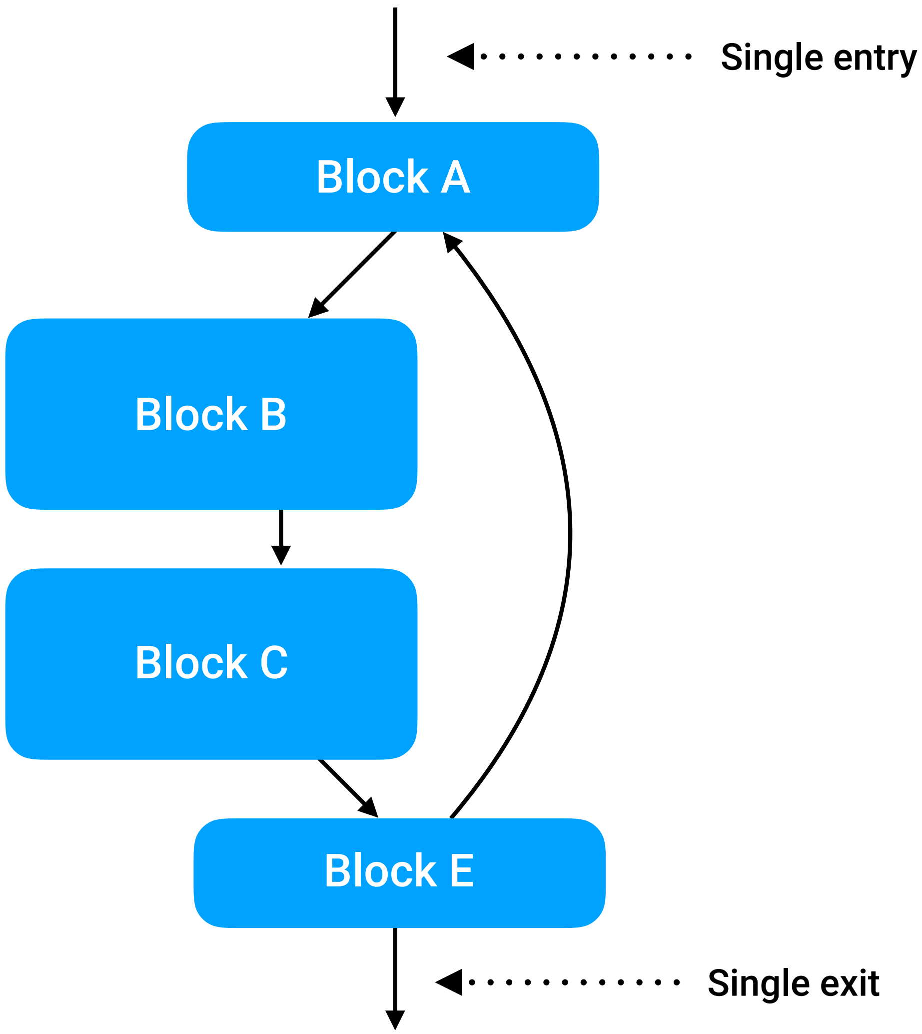
=> look into manual of your favourite compiler

Control Flow

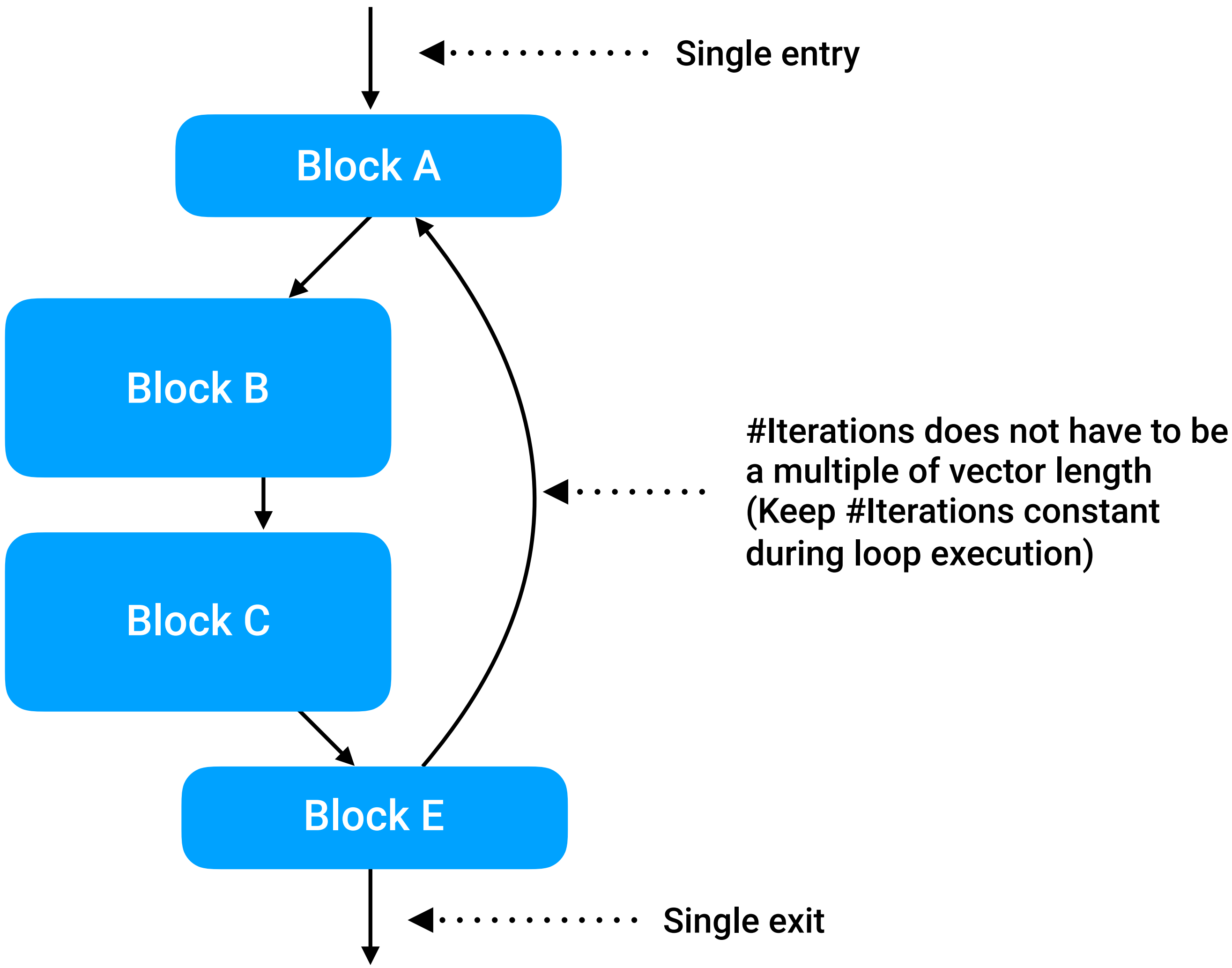
Control Flow



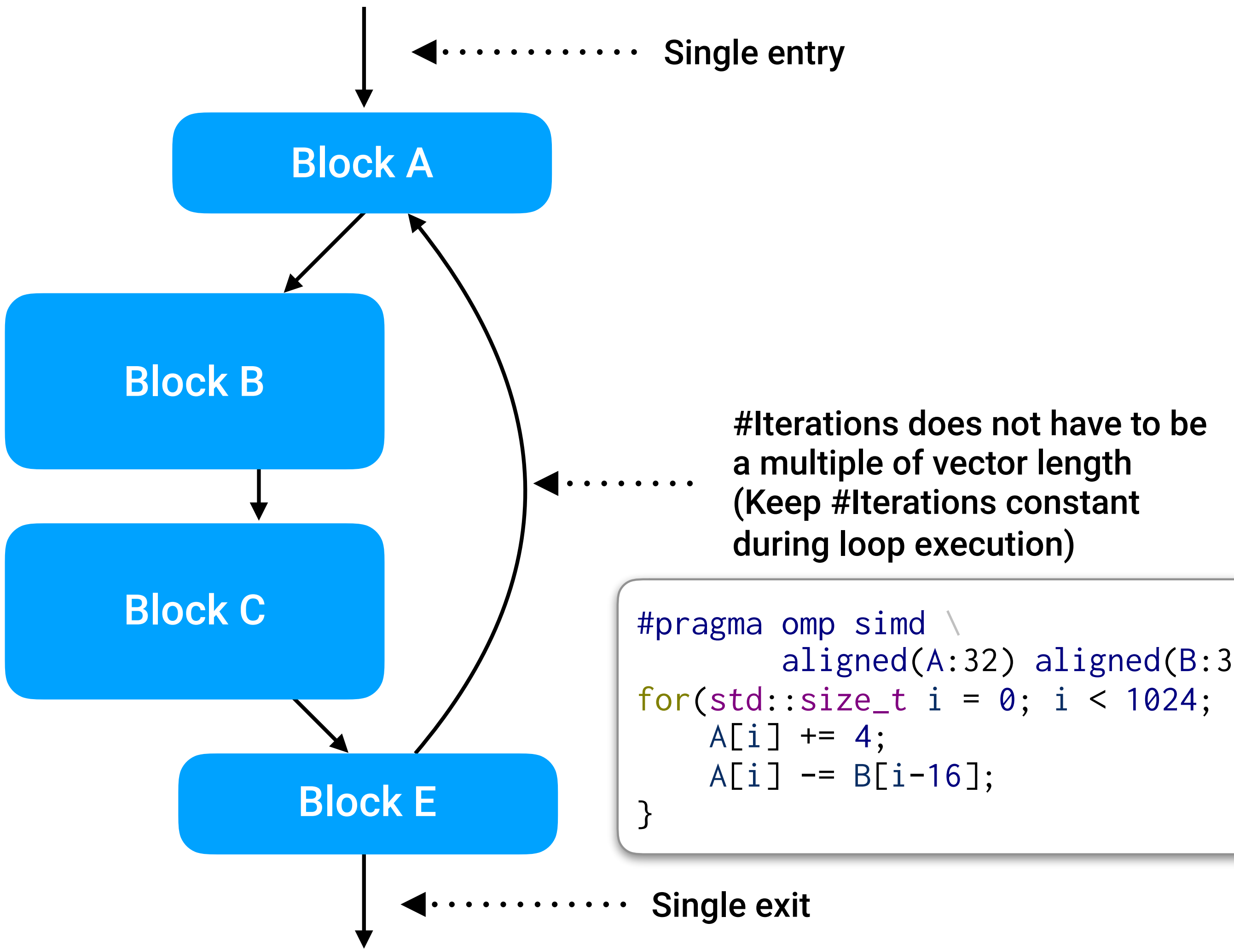
Control Flow



Control Flow

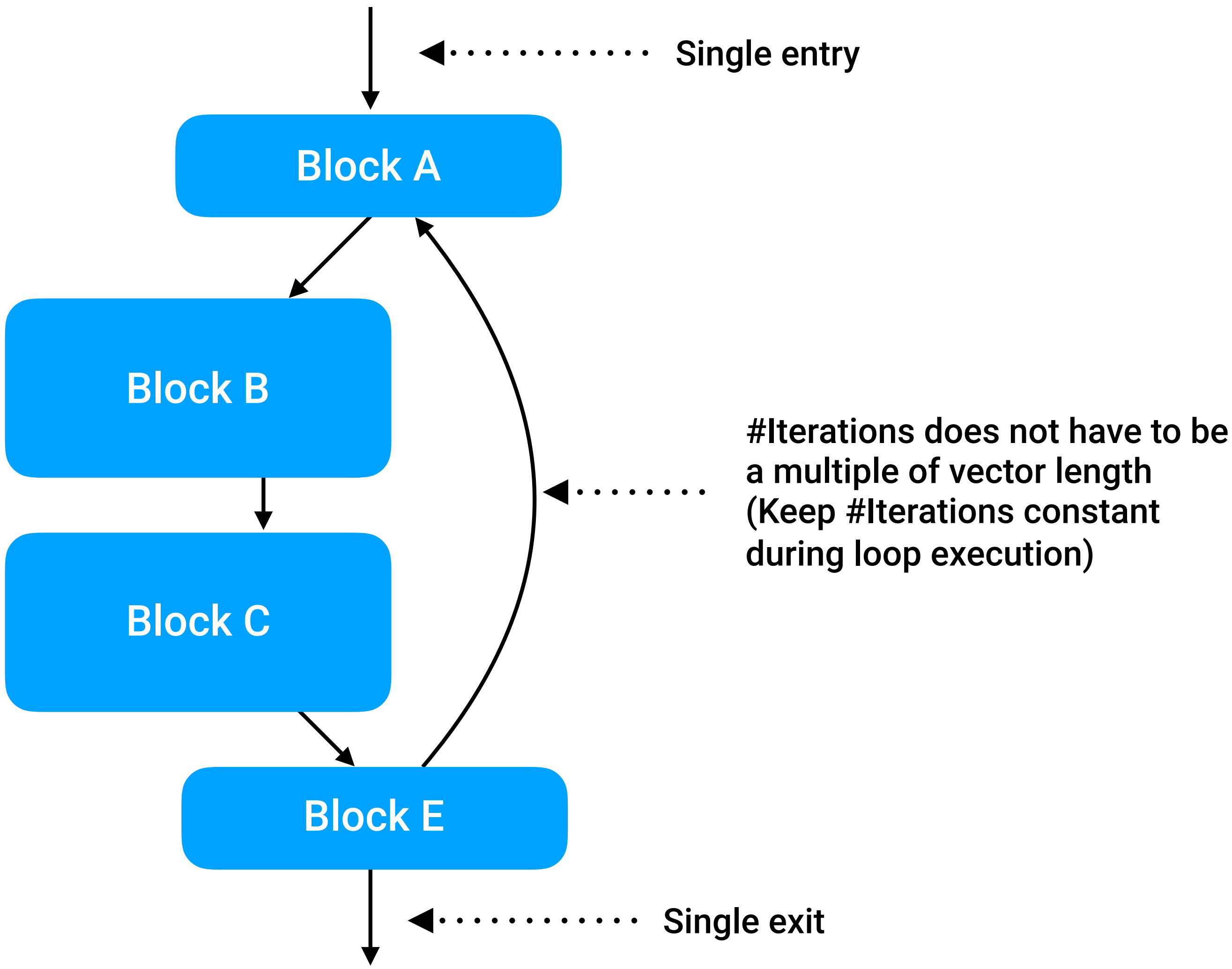


Control Flow

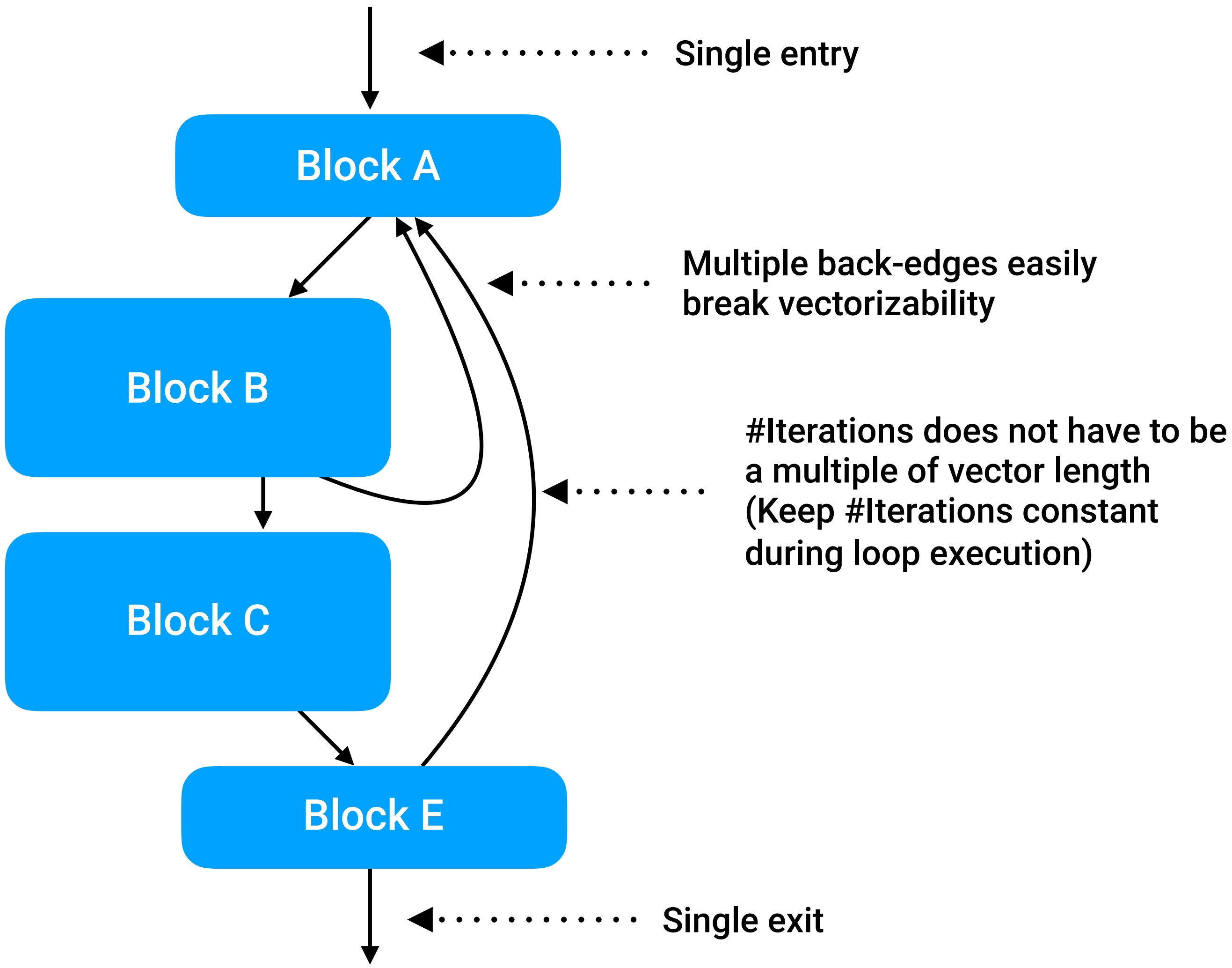



```
#pragma omp simd \
        aligned(A:32) aligned(B:32)
for(std::size_t i = 0; i < 1024; ++i) {
    A[i] += 4;
    A[i] -= B[i-16];
}
```

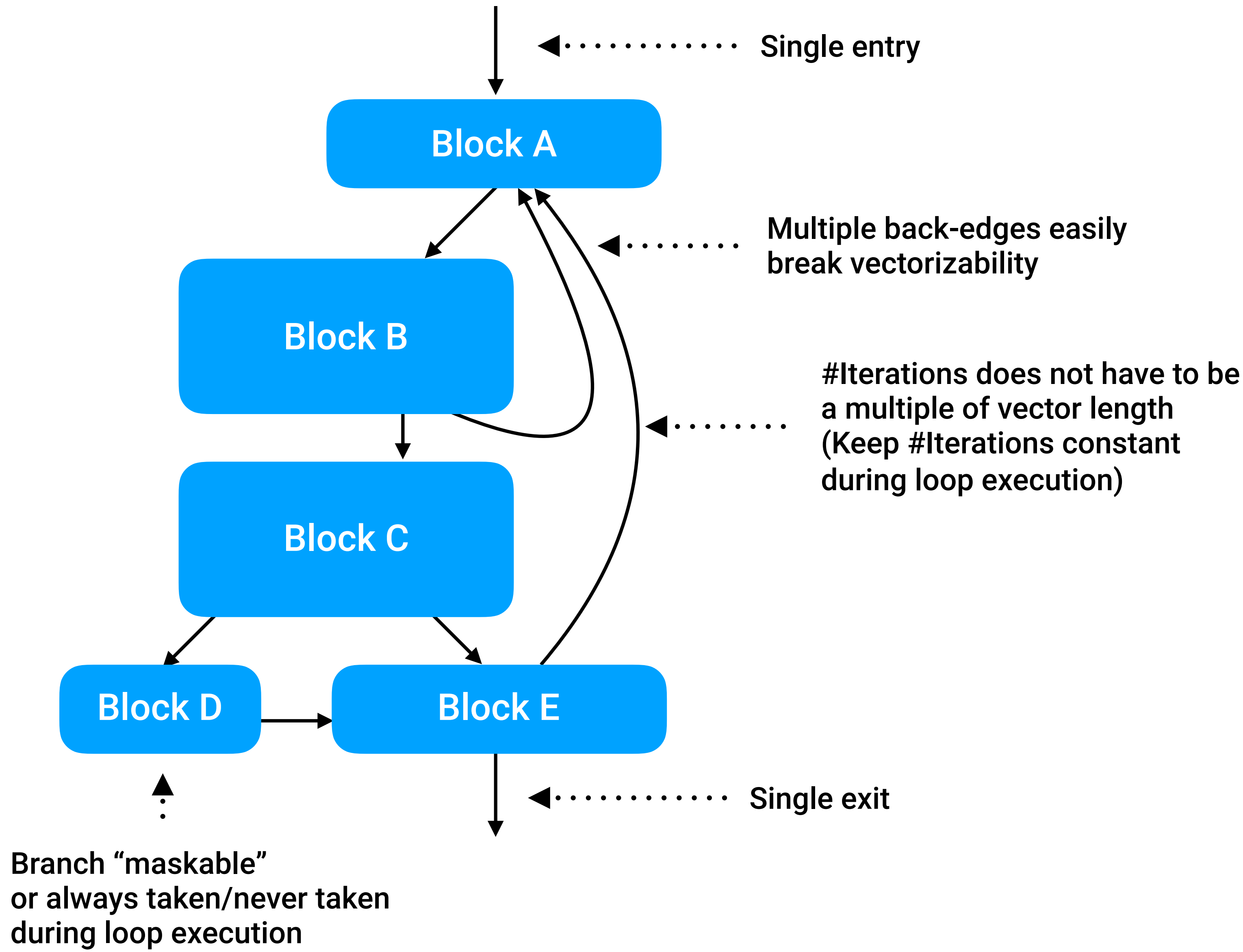
Control Flow



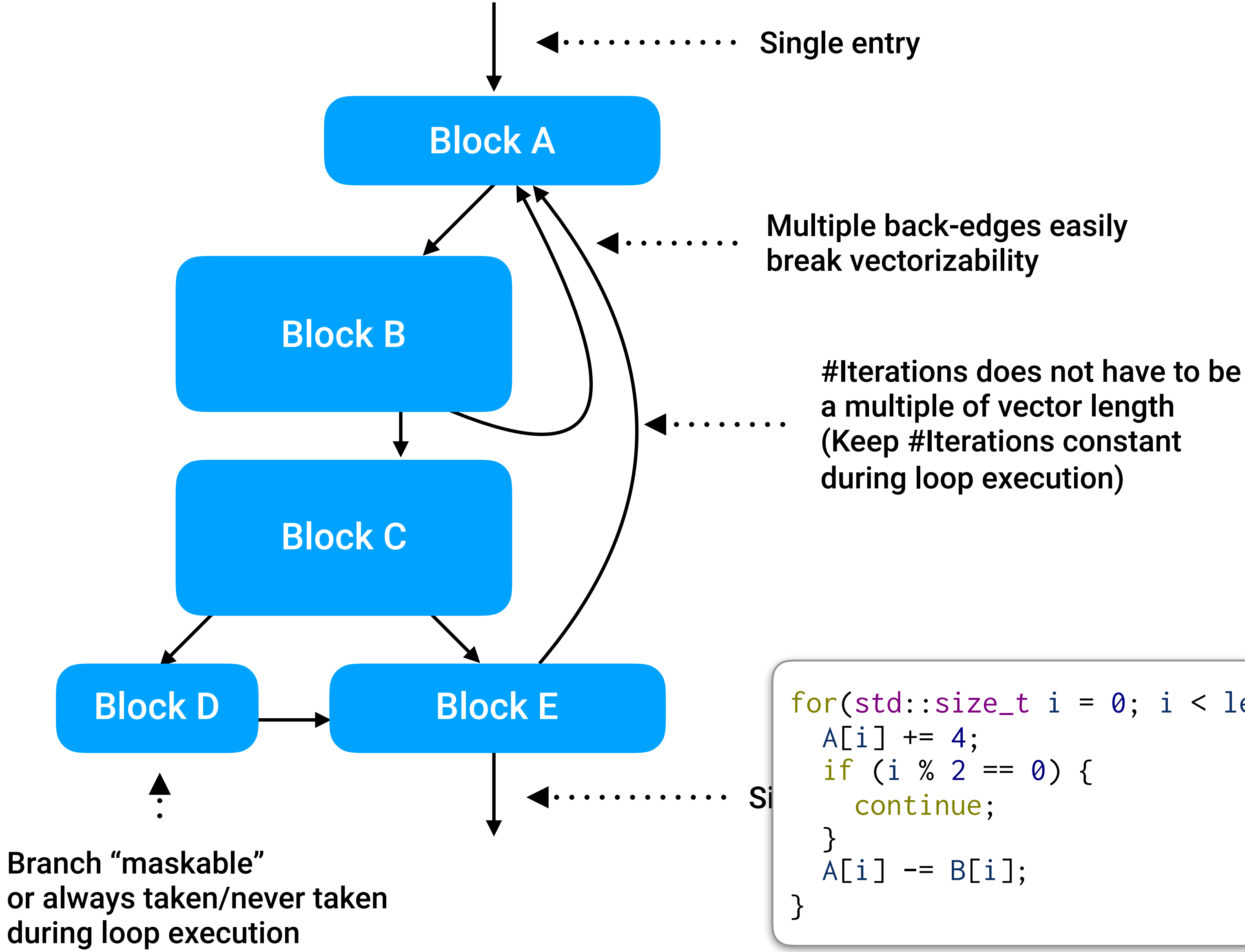
Control Flow



Control Flow



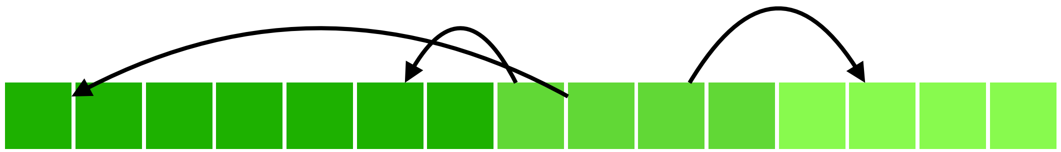
Control Flow



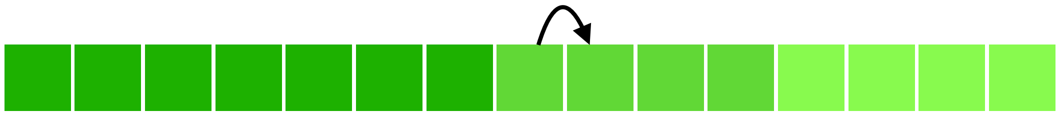

```
for(std::size_t i = 0; i < len; ++i) {
  A[i] += 4;
  if (i % 2 == 0) {
    continue;
  }
  A[i] -= B[i];
}
```

Data Dependencies Revisited

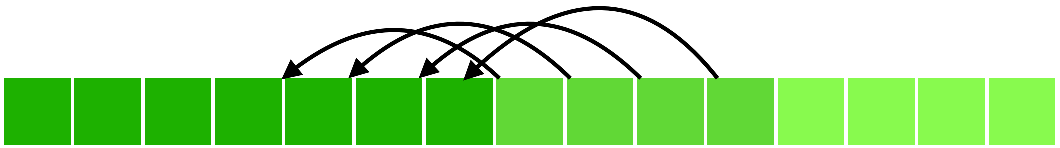
Hard to optimize:



Sometimes problematic:



Convenient:



Memory Access Stride

Memory Access Stride

```
float strided_add(float* src, float* target, int n) {
    for (int i = 0; i < n; ++i) {
        target[i] += src[4*i];
    }
    return 0.0f;
}
```

Memory Access Stride

```
float strided_add(float* src, float* target, int n) {
    for (int i = 0; i < n; ++i) {
        target[i] += src[4*i];
    }
    return 0.0f;
}
```

Memory Access Stride

```
float strided_add(float* src, float* target, int n) {
    for (int i = 0; i < n; ++i) {
        target[i] += src[4*i];
    }
    return 0.0f;
}
```

supported, but usually too costly -> rarely autovectorized

Data Dependencies

Data Dependencies

```
void read_after_write_bad(float* a, std::size_t n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n; ++i) {
        a[i] += a[i-1];
    }
}
```

Data Dependencies

Write-after-write wrt. vectors - not vectorizable

```
void read_after_write_bad(float* a, std::size_t n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n; ++i) {
        a[i] += a[i-1];
    }
}
```

Data Dependencies

Write-after-write wrt. vectors - not vectorizable

```
void read_after_write_bad(float* a, std::size_t n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n; ++i) {
        a[i] += a[i-1];
    }
}
```

```
void read_after_write_ok(float* a, std::size_t n) {
    #pragma omp simd safelen(16)
    for (std::size_t i = 16; i < n; ++i) {
        a[i] += a[i-16];
    }
}
```

Data Dependencies

Write-after-write wrt. vectors - not vectorizable

```
void read_after_write_bad(float* a, std::size_t n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n; ++i) {
        a[i] += a[i-1];
    }
}
```

```
void read_after_write_ok(float* a, std::size_t n) {
    #pragma omp simd safelen(16)
    for (std::size_t i = 16; i < n; ++i) {
        a[i] += a[i-16];
    }
}
```

write-after-read has analogous pitfalls

Alignment

Alignment

```
#pragma omp simd aligned(A:32) aligned(B:32)
for(std::size_t i = 0; i < n; ++i) {
    A[i] += 4;
    A[i] -= B[i-16];
}
```

Alignment

Aliasing

Aliasing

```
void add_to(float* A, float* B) {
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Aliasing

```
void add_to(float* A, float* B) {
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

```
void add_to(float* A, float* B) {
    #pragma omp simd
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Aliasing


```
void add_to(float* A, float* B) {
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

```
void add_to(float* A, float* B) {
    #pragma omp simd
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Aliasing


```
void add_to(float* A, float* B) {
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Autovectorized: safe

```
void add_to(float* A, float* B) {
    #pragma omp simd
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Aliasing


```
void add_to(float* A, float* B) {
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

Autovectorized: safe

```
void add_to(float* A, float* B) {
    #pragma omp simd
    for (std::size_t i = 0; i < 1024; ++i) {
        A[i] += B[i];
    }
}
```

potentially unsafe!

Aliasing

Aliasing

- C's `restrict` keyword missing in C++

Aliasing

- C's `restrict` keyword missing in C++
- clang: use `__restrict` to remove aliasing checks

Aliasing

- C’s `restrict` keyword missing in C++
- clang: use `__restrict` to remove aliasing checks
  - unfortunately, seems to prevent autovectorization by gcc

Reductions

```
int dot(int* A, int* B, int n) {
    int result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

Reductions

```
int dot(int* A, int* B, int n) {
    int result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

More than one reduction variable OK, too

Floating Point Operations

Floating Point Operations

```
int dot(float* A, float* B, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

Floating Point Operations

```
int dot(float* A, float* B, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

Vectorised version has different order of additions
- needs to be allowed explicitly:

Floating Point Operations

```
int dot(float* A, float* B, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

Vectorised version has different order of additions
- needs to be allowed explicitly:

```
#pragma omp simd
```

Floating Point Operations

```
int dot(float* A, float* B, int n) {
    float result = 0;
    for (int i = 0; i < n; ++i) {
        result += A[i] * B[i];
    }
    return result;
}
```

Vectorised version has different order of additions
- needs to be allowed explicitly:

```
#pragma omp simd
```

or

```
-ffast-math
```

Function Calls (Autovectorization)

Function Calls (Autovectorization)

- Called function may not throw

Function Calls (Autovectorization)

- Called function may not throw
- Autovectorization generally fails for non-inlined function calls

Function Calls (Autovectorization)

- Called function may not throw
- Autovectorization generally fails for non-inlined function calls
- Inlined code must be vectorizable

Function Calls (Autovectorization)

- Called function may not throw
- Autovectorization generally fails for non-inlined function calls
- Inlined code must be vectorizable
- clang/gcc: Might try `__attribute__((always_inline))`

Function Calls (OpenMP)

Function Calls (OpenMP)

```
#pragma omp declare simd uniform(z)
float scaled_avg(float lhs, float rhs, float z)
{
    return z*(lhs+rhs)/2.0f;
}
```

Function Calls (OpenMP)

z same for all calls
pertaining to a vector

```
#pragma omp declare simd uniform(z)
float scaled_avg(float lhs, float rhs, float z)
{
    return z*(lhs+rhs)/2.0f;
}
```

Function Calls (OpenMP)

Function Calls (OpenMP)

```
#pragma omp declare simd uniform(array) uniform(length) linear(index:1)
float stencil(float* array, int length, int index) {
    return 4.0f*array[index] - array[index-1] - array[index+1];
}

float stencil_all(float* __restrict a, float* __restrict b, int n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n-1; ++i) {
        b[i] = stencil(a, n, i);
    }

    return 0.0f;
}
```

Function Calls (OpenMP)

index increases with stride 1
during vector calls

```
#pragma omp declare simd uniform(array) uniform(length) linear(index:1)
float stencil(float* array, int length, int index) {
    return 4.0f*array[index] - array[index-1] - array[index+1];
}

float stencil_all(float* __restrict a, float* __restrict b, int n) {
    #pragma omp simd
    for (std::size_t i = 1; i < n-1; ++i) {
        b[i] = stencil(a, n, i);
    }

    return 0.0f;
}
```

Conditionals Revisited

Given

A =	a1	a2	a3	a4
B =	b1	b2	b3	b4

Compute

X =	min (a1,b1)	min (a2,b2)	min (a3,b3)	min (a4,b4)

Solution: Compute mask

M =	M1	M2	M3	M4

with Mi = 0xFFF...FFF if ai ≤ bi,
Mi = 0x000...000 otherwise

X = (A AND M) OR (B AND ~M)

Function Calls (OpenMP)

```
#pragma omp declare simd linear(a: 1) linear(b: 1) inbranch
float max(float* a, float* b) {
    if (*a > *b) {
        return *a;
    }
    return *b;
}
```

Function Calls (OpenMP)

Instructs the compiler to generate masking code

```
#pragma omp declare simd linear(a: 1) linear(b: 1) inbranch
float max(float* a, float* b) {
    if (*a > *b) {
        return *a;
    }
    return *b;
}
```

Pros & Cons: Vectorization

Pros & Cons: Vectorization

Benefits:

Pros & Cons: Vectorization

Benefits:

- Express code in C++

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent
- Compiler generates optimized code

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent
- Compiler generates optimized code

Drawbacks:

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent
- Compiler generates optimized code

Drawbacks:

- Fragility issues

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent
- Compiler generates optimized code

Drawbacks:

- Fragility issues
- Less control than with intrinsics

Pros & Cons: Vectorization

Benefits:

- Express code in C++
- (Micro)architecture-independent
- Compiler generates optimized code

Drawbacks:

- Fragility issues
- Less control than with intrinsics
- Vectorization capabilities different across compilers

Pros & Cons: Autovectorization

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

- Safety

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

- Safety
- Easy to debug

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

- Safety
- Easy to debug

Drawbacks:

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

- Safety
- Easy to debug

Drawbacks:

- No way to help the compiler with vectorization

Pros & Cons: Autovectorization

Benefits & drawbacks of vectorization, plus:

Benefits:

- Safety
- Easy to debug

Drawbacks:

- No way to help the compiler with vectorization
- Function calls need to be inlined

Pros & Cons: OpenMP

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops
- Can target GPGPUs

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops
- Can target GPGPUs
- Less dependent on heuristics than autovectorization

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops
- Can target GPGPUs
- Less dependent on heuristics than autovectorization

Drawbacks:

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops
- Can target GPGPUs
- Less dependent on heuristics than autovectorization

Drawbacks:

- Generated code not necessarily equivalent to scalar code

Pros & Cons: OpenMP

Benefits & drawbacks of vectorization, plus:

Benefits:

- Standardized
- Supports simultaneous vectorization and MIMD-parallelization of loops
- Can target GPGPUs
- Less dependent on heuristics than autovectorization

Drawbacks:

- Generated code not necessarily equivalent to scalar code
- Complex setups harder to debug

Conclusions

Conclusions

- SIMD can be beneficial on common CPUs

Conclusions

- SIMD can be beneficial on common CPUs
- Design algorithms & data structures with SIMD in mind

Conclusions

- SIMD can be beneficial on common CPUs
- Design algorithms & data structures with SIMD in mind
- Modern loop vectorizers succeed in many situations

Conclusions

- SIMD can be beneficial on common CPUs
- Design algorithms & data structures with SIMD in mind
- Modern loop vectorizers succeed in many situations
- First try vectorization; if that fails, use SIMD instructions more directly

Helpful Resources

Autovectorization in LLVM
https://llvm.org/docs/Vectorizers.html

Autovectorization in GCC
https://gcc.gnu.org/projects/tree-ssa/vectorization.html

OpenMP Application Programming Interface
http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf

OpenMP 4.0 API Quick Reference Card
http://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf

END