SIMD Programming and
What You Must Know about CPU Peak
FLOPS

Kenjiro Taura
Contents

1. Introduction

2. SIMD Instructions

3. SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4. What you must know to get near peak FLOPS
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
Remember performance of matrix-matrix multiply?

```c
void gemm(long n, /* n = 2400 */
    float A[n][n], float B[n][n], float C[n][n]) {
    long i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

$ ./simple_mm

C[1200][1200] = 3011.114014
in 56.382360 sec
2.451831 GFLOPS

$ ./opt_mm

C[1200][1200] = 3011.108154
in 1.302980 sec
106.095263 GFLOPS
Is matrix-matrix multiply important?

- it is used as a kernel of many computations
  - solving a system of linear equations ($Ax = b$)
  - it is in turn used in a partial differential equation solver
  - neural network
A kernel in Neural Network

- each layer of a neural network converts an $N$-vector $x$ to another $M$-vector $y$
- a general representation of a layer is a matrix-vector multiplication, followed by an element-wise non-linear function:
  \[ y = f(Wx) \]
- in training neural networks, this function is applied to a collection of vectors in a batch (mini batch)
  \[ Y = f(WX) \]
How much computation in NN?

- a typical natural language processing application:
  - $M = N = 512$ (the dimension of a word vector)
How much computation in NN?

- a typical natural language processing application:
  - $M = N = 512$ (the dimension of a word vector)
  - ⇒ for each word going through a layer, it takes $2MN \approx 5 \times 10^8$ flops

\[ y = f(Wx) \]
How much computation in NN?

- a typical natural language processing application:
  - \( M = N = 512 \) (the dimension of a word vector)
  - \( \Rightarrow \) for each word going through a layer, it takes \( 2MN \approx 5 \times 10^8 \) flops

\[
y = f(Wx)
\]

- \( \Rightarrow \) each sentence has a number of words (\( \approx 20 \)); it takes proportionally many flops.

\[
(5 \times 10^8) \times 20 = 10^{10} \text{ flops}
\]
How much computation in NN?

- a typical natural language processing application:
  - $M = N = 512$ (the dimension of a word vector)
  - for each word going through a layer, it takes
    $2MN \approx 5 \times 10^8$ flops

$$y = f(Wx)$$

- each sentence has a number of words ($\approx 20$); it takes proportionally many flops.

$$(5 \times 10^8) \times 20 = 10^{10} \text{ flops}$$

- training the network with a million (1M) sentences requires:

$$10^{10} \times 10^6 = 10^{16} \text{ flops}$$
How much computation in NN?

- a typical natural language processing application:
  - $M = N = 512$ (the dimension of a word vector)
  - for each word going through a layer, it takes $2MN \approx 5 \times 10^8$ flops

\[
y = f(Wx)
\]

- each sentence has a number of words ($\approx 20$); it takes proportionally many flops.

\[
(5 \times 10^8) \times 20 = 10^{10} \text{ flops}
\]

- training the network with a million (1M) sentences requires:

\[
10^{10} \times 10^6 = 10^{16} \text{ flops}
\]

- with 10 GFLOPS ($= 10^{10}$ flops/sec), it takes $10^6$ seconds or 11 days
A note for NLP experts

- the actual amount of computation easily becomes an order of magnitude more
- $10^{16}$ flops is just for the *forwarding* phase, with a *single* weight matrix
- NLP applications typically use Long Short Term Memory (LSTM), which takes *eight* such matrices in each layer ($\times 8$)
- a back propagation takes roughly the same amount of flops ($\times 2$)
What is the theoretical limit?

- we are using Intel Haswell processor
- its *single core* can execute, in *every cycle*,
  - two *fused multiply-add instructions*
  - and others (e.g., integer arithmetic, load, store, ...) I’ll cover later
- moreover, a single fused multiply-add *instruction* can multiply/add *four* double-precision or *eight* single-precision operands
- *Single Instruction Multiple Data (SIMD) instructions*
Terminology

- **flops**: floating point operations
- **FLOPS**: Floating Point Operations Per Second
- Practically,

  \[
  \text{Peak FLOPS} = \text{FP operands per instruction (SIMD width)} \times \text{instructions per cycle (IPC)} \times \text{cycles per second (frequency)}
  \]
Recent processors increasingly rely on SIMD as an energy efficient way for boosting peak FLOPS

<table>
<thead>
<tr>
<th>Microarchitecture</th>
<th>ISA</th>
<th>throughput (per clock)</th>
<th>SIMD width</th>
<th>max flops/cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nehalem</td>
<td>SSE</td>
<td>1 add + 1 mul</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Sandy Bridge</td>
<td>AVX</td>
<td>1 add + 1 mul</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>/Ivy Bridge</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Haswell</td>
<td>AVX2</td>
<td>2 fmadds</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Knights Corner</td>
<td>AVX-512F</td>
<td>1 fmadd(∗)</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Knights Landing</td>
<td>AVX-512F</td>
<td>2 fmadds</td>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

- **SIMD width**: number of *single precision* operands
- **ISA**: Instruction Set Architecture
- **fmadd**: fused multiply-add
- **∗**: single hardware thread can achieve half of it
Today’s theme

- understand how/when you can approach CPU’s performance limit
- we’ll learn
  - practical ways to use SIMD instructions
  - basics of processors to know what kind of code can get close-to-peak performance
Contents

1. Introduction

2. SIMD Instructions

3. SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4. What you must know to get near peak FLOPS
Intel SIMD instructions at a glance

Some example AVX instructions

<table>
<thead>
<tr>
<th>operation</th>
<th>syntax</th>
<th>C-like expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiply</td>
<td><code>vmlups %ymm0,%ymm1,%ymm2</code></td>
<td><code>ymm2 = ymm1 * ymm0</code></td>
</tr>
<tr>
<td>add</td>
<td><code>vaddps %ymm0,%ymm1,%ymm2</code></td>
<td><code>ymm2 = ymm1 + ymm0</code></td>
</tr>
<tr>
<td>fmadd</td>
<td><code>vfmadd132ps %ymm0,%ymm1,%ymm2</code></td>
<td><code>ymm2 = ymm0*ymm2+ymm1</code></td>
</tr>
<tr>
<td>load</td>
<td><code>vmovaps 400(%rax),%ymm0</code></td>
<td><code>ymm0 = *(rax+400)</code></td>
</tr>
<tr>
<td>store</td>
<td><code>vmovaps %ymm0,400(%rax)</code></td>
<td><code>*(rax+400) = ymm0</code></td>
</tr>
</tbody>
</table>

- `ymm0` ... `ymm15` are 256 bit registers
  - can hold eight single precision (`float` of C) or four double precision (`double` of C) floating point numbers
- `XXXXps` stands for `packed single precision`
### Intel SIMD instructions at a glance

Don’t be confused by similar but different variations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Vector/Scalar?</th>
<th>Width (bits)</th>
<th>ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vmulps %ymm0,%ymm1,%ymm2</code></td>
<td>8 SPs</td>
<td>vector</td>
<td>256</td>
<td>AVX</td>
</tr>
<tr>
<td><code>vmulpd %ymm0,%ymm1,%ymm2</code></td>
<td>4 DPs</td>
<td>vector</td>
<td>256</td>
<td>AVX</td>
</tr>
<tr>
<td><code>mulps %xmm0,%xmm1</code></td>
<td>4 SPs</td>
<td>vector</td>
<td>128</td>
<td>SSE</td>
</tr>
<tr>
<td><code>mulpd %xmm0,%xmm1</code></td>
<td>2 DPs</td>
<td>vector</td>
<td>128</td>
<td>SSE</td>
</tr>
<tr>
<td><code>mulss %xmm0,%xmm1</code></td>
<td>1 SP</td>
<td>scalar</td>
<td>(32)</td>
<td>SSE</td>
</tr>
<tr>
<td><code>mulsd %xmm0,%xmm1</code></td>
<td>1 DP</td>
<td>scalar</td>
<td>(64)</td>
<td>SSE</td>
</tr>
<tr>
<td><code>vfmadd132ss %ymm0,%ymm1,%ymm2</code></td>
<td>1 SP</td>
<td>scalar</td>
<td>(32)</td>
<td>AVX</td>
</tr>
</tbody>
</table>

- ... ps: packed *single* precision
- ... pd: packed *double* precision
- xmm0, ..., xmm15: 128 bit SSE registers (aliased to lower half of ymm registers)
- Intel instructions have traditionally been taking two operands, but since AVX, many new instructions become three operands
Applications/limitations of SIMD

- SIMD is good at parallelizing computations doing *almost exactly* the same series of instructions on contiguous data

  ⇒ generally, main targets are simple loops whose iterations have identical control flows

- less effective for
  - loops with conditional branches
  - loops with nested loops of variable trip counts
  - loops with unknown function calls
  - loops with non-contiguous memory accesses
  - non loops
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
Alternative ways to use SIMD

- auto vectorization
  - loop vectorization
  - basic block vectorization
- explicit SIMD constructs/directives
  - SIMD directives for loops (OpenMP 4.0/Cilk Plus)
  - SIMD-enabled functions (OpenMP 4.0/Cilk Plus)
  - array languages (Cilk Plus)
  - specially designed languages
- somewhat portable vector types
  - GCC vector extensions
  - Boost.SIMD
- intrinsics
- assembly programming
the following slides illustrate how you vectorize the following simple operation

\[ x = ax + c \]

where \( x \) is an array; and \( a \) and \( c \) scalars

```c
void axpy(float a, float * x, float c, long m) {
  long j;
  for (j = 0; j < m; j++) {
    x[j] = a * x[j] + c;
  }
}
```
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
Auto loop vectorization

- write scalar loops and hope the compiler does the job
- e.g.,

```c
void axpy_auto(float a, float * x, float c, long m) {
    long j;
    for (j = 0; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}
```

- compile and run

```bash
$ gcc -o simd_auto -march=native -O3 simd_auto.c
$ master:6simd% ./simd_auto
8000000000.000000 flops
1226107900 clocks
6.524711 flops/clock
```
How to know if the compiler vectorized it?

- there are options useful to know why it didn’t vectorize

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Report successful vectorization</th>
<th>Report failed vectorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC &lt; 4.8</td>
<td>-ftree-vectorizer-verbose=1</td>
<td>-ftree-vectorizer-verbose=2</td>
</tr>
<tr>
<td></td>
<td>-fopt-info-vec-optimized</td>
<td>-fopt-info-vec-missed</td>
</tr>
<tr>
<td>GCC ≥ 4.8</td>
<td>-vec-report=1</td>
<td>-vec-report=2</td>
</tr>
<tr>
<td>Intel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- but don’t hesitate to dive into assembly code

- gcc -S is your friend
- trick: *enclose loops with inline assembler comments*

```c
1  asm volatile("# xxx loop begin");
2  for (i = 0; i < n; i++) {
3      ... /* hope to be vectorized */
4  }
5  asm volatile("# xxx loop end");
```
Auto vectorization fails easily

- potential aliasing
- misaligned data make vectorization less efficient
- complex control flows make vectorization less efficient

_Telling the compiler “they don’t happen; don’t worry about them” helps compilers do a better job_
“auto” vectorizer must guarantees that a vectorized version produces an identical result with a non-vectorized version.

Loops operating on two or more arrays are often not vectorizable if they happen to be the same array.

```c
for (i = 0; i < m; i++) {
    y[i] = a * x[i] + c;
}
```

good compilers generate code that first checks $x[i:i+8]$ and $y[i:i+8]$ partially overlap.

```
$ gcc -march=native -O3 -fopt-info-vec-optimized
...
simd_auto.c:30:3: note: loop versioned for vectorization because of possible aliasing
...
```

If you know they don’t, it’s advisable to make that explicit.

`restrict` keyword, introduced by C99, does just that.
restrict keyword

- annotate parameters of pointer type with `restrict`, if you know they never point to the same data

```c
void axpy(float a, float * restrict x, float c,
          float * restrict y, long m) {
    for (long j = 0; j < m; j++) {
        y[j] = a * x[j] + c;
    }
}
```

- you need to specify `-std=c99` or `-std=gnu99`

```sh
$ gcc -march=native -O3 -S a.c -std=gnu99 -fopt-info-vec-optimized
... 
a.c:5: note: LOOP VECTORIZED.
a.c:1: note: vectorized 1 loops in function.
... 
```
Alignment and vectorization

- background: SIMD load/store instructions require/prefer the address to be aligned to vector size
  - movaps: requires 16 byte-aligned address
  - vmovaps: requires 32 byte-aligned address

- even for loops as simple as this

  ```
  1 for (j = 0; j < m; j++) {
  2     x[j] = a * x[j] + c;
  3  }
  ```

  the generated code runs a few scalar iterations until the address of x[j] becomes aligned

- if possible,
  - (step 1): align your data appropriately and
  - (step 2): let the compiler know about it
Step 1: align data

- static/local/global variables and arrays
  
  ```c
  float a[100] __attribute__((aligned(64)));
  ```

- dynamically allocated data
  
  ```c
  float * a;
  int err = posix_memalign(&a, 64, sizeof(float) * 100);
  if (err) failed();
  ```
Step 2: let the compiler know about alignment

- GCC (≥ 4.6)
  ```c
  x = __builtin_assume_aligned(x, 64);
  ```
  returns x, except the compiler assumes it is now 64 byte-aligned

- GCC (< 4.6); only works for function parameters
  ```c
  void f(float * __attribute__((aligned(64))) x) {
      ...
  }
  ```
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
**SIMD constructs**

- **simd pragma**
  - allows an explicit vectorization of for loops
  - syntax restrictions similar to `omp for` pragma apply

- **declare simd pragma**
  - instructs the compiler to generate vectorized versions of a function
  - with it, loops with function calls can be vectorized
**simd pragma**

- **basic syntax** (similar to `omp for`):

```c
#pragma omp simd clauses
for (i = ...; i < ...; i += ...)
  S
```

- **clauses**
  - `aligned(var,var,...:align)`
  - `uniform(var,var,...)` says variables are loop invariant
  - `linear(var,var,...:stride)` says variables have the specified stride between consecutive iterations
void axpy_omp(float a, float * x, float c, long m) {
    long j;
    #pragma omp simd
    for (j = 0; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}

- note: there are no points in using `omp simd` here, when auto vectorization does the job
- in general, `omp simd` declares “you don’t mind that the vectorized version is not the same as non-vectorized version”
**simd** pragma to vectorize programs explicitly

- computing an inner product:

```c
void inner_omp(float * x, float * y, long m) {
    long j;
    float c = 0;
    #pragma omp simd reduction(c:+)
    for (j = 0; j < m; j++) {
        c += x[j] * y[j];
    }
}
```

- note that the above loop is unlikely to be auto-vectorized, due to dependency through `c`
you can vectorize a function body, so that it can be called within a vectorized context

basic syntax (similar toomp for):

1. `#pragma omp declare simd clauses`
2. `function definition`

clauses
- those for `simd pragma`
- `notinbranch`
- `inbranch`

GCC’s vectorized does not do a decent job, as of now
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
GCC vector types

- GCC allows you to define a vector type
  ```c
  typedef float float8 __attribute__((vector_size(32)));  
  ```

- You can use arithmetic on vector types
  ```c
  float8 x, y, z;
  z += x * y;
  ```

- You cannot mix scalar and vectors, however
  ```c
  float8 x, y, z;
  z = 3.5 * x + y;
  ```

- You can combine them with intrinsics
axpy in GCC vector extension

scalar code

```c
void axpy_auto(float a, float * x, float c, long m) {
    long j;
    for (j = 0; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}
```

GCC vector extension

```c
typedef float float8 __attribute__((vector_size(32)));

void axpy_vector_ext(float a, float * x, float c, long m) {
    long j;
    float8 a8 = { a,a,a,a,a,a,a,a };  // a8 = { a, a, a, a, a, a, a, a }
    float8 c8 = { c,c,c,c,c,c,c,c };  // c8 = { c, c, c, c, c, c, c, c }
    for (j = 0; j + 7 < m; j += 8) {
        *((float8*)&x[j]) = a8 * *((float8*)&x[j]) + c8;
    }
    for (; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}
```
Contents

1 Introduction

2 SIMD Instructions

3 SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4 What you must know to get near peak FLOPS
Vector intrinsics

- processor/platform-specific functions and types
- on x86 processors, put this in your code

```c
#include <x86intrin.h>
```

and you get

- a set of available vector types
- a lot of functions operating on vector types

Vector intrinsics

- vector types:
  - __m128 (128 bit vector),
  - __m256 (256 bit vector),
  - ...

- functions operating on vector types:
  - __mm_xxx (128 bit),
  - __mm256_xxx (256 bit),
  - ...

- each function almost directly maps to a single assembly instruction, so you know what will happen
  - __mm_add_ps, etc.
  - __mm256_load_ps
  - __mm256_store_ps
  - __mm256_add_ps
```c
void axpy_intrin(float a, float * x, float c, long m) {
    long j;
    for (j = 0; j < m; j += 8) {
        _mm256_store_ps(&x[j],
            _mm256_add_ps(_mm256_mul_ps(_mm256_set1_ps(a),
                _mm256_load_ps(&x[j])),
            _mm256_set1_ps(c)));
    }
}
```

- ugly
- you would like to use vector extension (or trivially vectorizable loops) and use intrinsics sparingly only when necessary
  - _mm256_set1_ps(x): extends a scalar to a vector
  - _mm256_shuffle_ps(v): shuffle words in a vector
  - etc.
a somewhat cleaner axpy with vector intrinsics + vectors

typedef float float8 __attribute__((vector_size(32)));

void axpy_vector_ext(float a, float * x, float c, long m) {
    long j;
    float8 a8 = _mm256_set1_ps(a);
    float8 c8 = _mm256_set1_ps(c);
    for (j = 0; j + 7 < m; j += 8) {
        *((float8*)&x[j]) = a8 * *((float8*)&x[j]) + c8;
    }
    for (; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}
1. Introduction

2. SIMD Instructions

3. SIMD programming alternatives
   - Auto loop vectorization
   - OpenMP SIMD Directives
   - GCC’s Vector Types
   - Vector intrinsics

4. What you must know to get near peak FLOPS
so you now know how to use SIMD instructions, which may bring performance boost up to the number of SIMD lanes
axpy, inner product, and matrix multiply, all seem particularly well-suited to SIMD
yet, just vectorizing code does not automatically bring you a satisfactory result
An endeavor to nearly peak FLOPS

- recall the inner-product that successfully vectorized

```c
float inner_omp(float * x, float * y, long m) {
    long j;
    float s = 0;
    #pragma omp simd reduction(+:s)
    for (j = 0; j < m; j++) {
        s += x[j] * y[j];
    }
    return s;
}
```

when run, we got:

```
$ ./inner_omp
40000000000.000000 flops
13069902676 clocks
3.060467 flops/clock
```

- mere 10% of the peak (32 flops/clock)
- what’s going on?
An endeavor to nearly peak FLOPS

- let’s further simplify the code, so that it does not do anything other than multiply and add
- can it be simpler than this?

```c
typedef float float8 __attribute__((vector_size(32)));

float8 axpy_1(float8 a, float8 x, float8 c, long n) {
    long i;
    for (i = 0; i < n; i++) {
        x = a * x + c;
    }
    return x;
}
```

- the code performs $16n$ flops
- no memory access in the loop
Let’s run it!

- compile

```
$ gcc -o axpy -march=native -O3 axpy.c
```

and run!

```
$ ./axpy
1600000000.000000 flops
501376128 clocks
3.191217 flops/clock
```

still far from the peak
Let’s run it!

compile

```
$ gcc -o axpy -march=native  -O3 axpy.c
```

and run!

```
$ ./axpy
1600000000.000000 flops
501376128 clocks
3.191217 flops/clock
```
Let’s run it!

- compile

```
1 $ gcc -o axpy -march=native -O3 axpy.c
```

- and run!

```
1 $ ./axpy
2 1600000000.000000 flops
3 501376128 clocks
4 3.191217 flops/clock
```

- still far from the peak
How to investigate

- put a landmark in the assembly code

```c
float8 axpy_1(float8 a, float8 x, float8 c, long n) {
    long i;
    asm volatile ("# ax+c loop begin");
    for (i = 0; i < n; i++) {
        x = a * x + c;
    }
    asm volatile ("# ax+c loop end");
    return x;
}
```
How to investigate

- put a landmark in the assembly code

```c
float8 axpy_1(float8 a, float8 x, float8 c, long n) {
    long i;
    asm volatile ("# ax+c loop begin");
    for (i = 0; i < n; i++) {
        x = a * x + c;
    }
    asm volatile ("# ax+c loop end");
    return x;
}
```

- compile into assembly

```bash
$ gcc -S -march=native -O3 axpy.c
```

- and see `axpy.s` in your editor
Assembly

```assembly
# ax+c loop begin
# 0 "" 2
#NO_APP
testq %rdi, %rdi
jle .L2
xorl %eax, %eax
.p2align 4,,10
.p2align 3
.L3:
addq $1, %rax
vfmadd132ps %ymm0, %ymm2, %ymm1
cmpq %rax, %rdi
jne .L3
.L2:
#APP
# 21 "axpy.c" 1
# ax+c loop end
```

- a little calculation shows the loop is taking
  \((16/3.191217) \approx 5 \text{ cycles per iteration}\)

- why?
Suspect looping overhead?

- if you suspect looping overhead, here is an unrolled version, with almost no looping overhead

- its performance is identical

```c
float8 axpy_2(float8 a, float8 x, float8 c, long n) {
    long i;
    asm volatile("# ax+c loop 2 begin");
    for (i = 0; i < n; i += 4) {
        x = a * x + c;
        x = a * x + c;
        x = a * x + c;
        x = a * x + c;
    }
    asm volatile("# ax+c loop 2 end");
    return x;
}
```
Latency and throughput

- Haswell core can execute two vfmaddps instructions every cycle
- yet, it does not mean the result of vfmaddps at line 3 is available in the next cycle for vfmaddps at line 4

```assembly
.L9:
    addq  $4, %rax
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    cmpq  %rax, %rdi
    jg .L9
```
Latency and throughput

- Haswell core can execute two \texttt{vfmaddps} instructions every cycle
- yet, it does \textit{not} mean the result of \texttt{vfmaddps} at line 3 is available in the next cycle for \texttt{vfmaddps} at line 4

```assembly
.L9:
1    addq   $4, %rax
2    vfmadd132ps %ymm0, %ymm2, %ymm1
3    vfmadd132ps %ymm0, %ymm2, %ymm1
4    vfmadd132ps %ymm0, %ymm2, %ymm1
5    vfmadd132ps %ymm0, %ymm2, %ymm1
6    vfmadd132ps %ymm0, %ymm2, %ymm1
7    cmpq   %rax, %rdi
8    jg    .L9
```

- \textit{what you need to know:}
  - “two \texttt{vfmadd132ps} instructions every cycle” refers to the \textit{throughput}
  - each instruction has a specific \textit{latency}
Haswell Latencies

<table>
<thead>
<tr>
<th>instruction</th>
<th>latency (cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fp add</td>
<td>3</td>
</tr>
<tr>
<td>fp mul</td>
<td>5/3 (*)</td>
</tr>
<tr>
<td>fp fmadd</td>
<td>5</td>
</tr>
<tr>
<td>typical integer ops</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(*): 5 on Haswell / 3 on Broadwell

- [http://www.agner.org/optimize/](http://www.agner.org/optimize/) is an invaluable resource
- 3. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers
- 4. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs
Our code in light of latencies

- in our code, a \texttt{vfmadd} uses the result of the immediately preceding \texttt{vfmadd}
- that was obvious from the source code too

```assembly
.L9:
    addq    $4, %rax
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    vfmadd132ps %ymm0, %ymm2, %ymm1
    cmpq    %rax, %rdi
    jg .L9
```

```
for (i = 0; i < n; i += 4) {
    x = a * x + c;
    x = a * x + c;
    x = a * x + c;
    x = a * x + c;
}
```

Conclusion:

*the loop can’t run faster than 5 cycles/iteration*
Variants

1. add only:

   ```
   for (i = 0; i < n; i += 4) {
     x = x + c;
     ...
   }
   ```

2. no fmadd (separate add + mul)

   ```
   gcc -mavx
   ```
   
   (fmadd isn’t used unless you give -march=native or -mfma)

3. mul only (watch for it!):

   ```
   for (i = 0; i < n; i += 4) {
     x = a * x;
     ...
   }
   ```

<table>
<thead>
<tr>
<th>variant</th>
<th>clocks / iter</th>
</tr>
</thead>
<tbody>
<tr>
<td>add only</td>
<td>3.0027</td>
</tr>
<tr>
<td>no fmadd</td>
<td>6.0070 (*)</td>
</tr>
<tr>
<td>mul only</td>
<td>127.1099</td>
</tr>
</tbody>
</table>

(*) The experiment was conducted on Broadwell, which has a multiply latency = 3
How to overcome latencies?

- increase parallelism!
How to overcome latencies?

- increase parallelism!
- you *can’t* make a serial chain of computation faster (change the algorithm if you want to)
How to overcome latencies?

- increase parallelism!
- you can’t make a serial chain of computation faster (change the algorithm if you want to)
- you can only increase throughput, by running many independent chains
How to overcome latencies?

- increase parallelism!
- you *can’t* make a serial chain of computation faster (change the algorithm if you want to)
- you *can* only increase *throughput*, by running many independent chains
- we expect the following to finish in the same number of cycles as the original one, despite it performs twice as many flops

```c
for (i = 0; i < n; i++) {
    x0 = a * x0 + c;
    x1 = a * x1 + c;
}
```
How to overcome latencies?

- increase parallelism!
- you *can’t* make a serial chain of computation faster (change the algorithm if you want to)
- you *can* only increase *throughput*, by running many independent chains
- we expect the following to finish in the same number of cycles as the original one, despite it performs twice as many flops
- note that two series are *independent*

```plaintext
for (i = 0; i < n; i++) {
    x0 = a * x0 + c;
    x1 = a * x1 + c;
}
```
...and you can have more than two chains ...

- we expect to reach peak FLOPS with $\geq 10$ chains and we do

```c
float8 axpy(float8 a, float8 x0, float8 x1, ...
            ...", float8 c, long n) {
    for (i = 0; i < n; i++) {
        x0 = a * x0 + c;
        x1 = a * x1 + c;
        ...
    }
    return x0 + x1 + ...;
}
```

<table>
<thead>
<tr>
<th>chains</th>
<th>clocks/iter</th>
<th>flops/clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.02</td>
<td>3.19</td>
</tr>
<tr>
<td>2</td>
<td>5.02</td>
<td>6.38</td>
</tr>
<tr>
<td>3</td>
<td>5.03</td>
<td>9.53</td>
</tr>
<tr>
<td>4</td>
<td>5.04</td>
<td>12.71</td>
</tr>
<tr>
<td>6</td>
<td>5.05</td>
<td>19.03</td>
</tr>
<tr>
<td>8</td>
<td>5.04</td>
<td>25.4</td>
</tr>
<tr>
<td>10</td>
<td>5.11</td>
<td>31.36</td>
</tr>
<tr>
<td>11</td>
<td>5.65</td>
<td>31.19</td>
</tr>
<tr>
<td>12</td>
<td>6.1</td>
<td>31.5</td>
</tr>
<tr>
<td>13</td>
<td>6.59</td>
<td>31.58</td>
</tr>
<tr>
<td>14</td>
<td>7.11</td>
<td>31.51</td>
</tr>
<tr>
<td>15</td>
<td>12.34</td>
<td>19.45</td>
</tr>
<tr>
<td>16</td>
<td>12.11</td>
<td>21.14</td>
</tr>
<tr>
<td>17</td>
<td>12.22</td>
<td>22.27</td>
</tr>
<tr>
<td>18</td>
<td>40.07</td>
<td>7.24</td>
</tr>
</tbody>
</table>
Superscalar processors

What you need to know:
Superscalar processors

What you need to know:

- instructions are decoded in the order of the program,
Superscalar processors

What you need to know:

- instructions are decoded in the order of the program,
- but execution orders are not constrained by it (*out of order execution*)
Superscalar processors

What you need to know:

- instructions are decoded in the order of the program,
- but execution orders are not constrained by it \textit{(out of order execution)}
- \(\Rightarrow\) as a first-order approximation, performance is constrained only by

\textit{latency:} length of dependent chain of instructions

\textit{throughput:} the number of particular types of instructions it can execute per cycle (e.g., two fmadds/cycle)
Superscalar processors

What you need to know:

- instructions are decoded in the order of the program,
- but execution orders are not constrained by it *out of order execution*
- ⇒ as a first-order approximation, performance is constrained only by
  - *latency*: length of dependent chain of instructions
Superscalar processors

What you need to know:

- instructions are decoded in the order of the program,
- but execution orders are not constrained by it (out of order execution)
- ⇒ as a first-order approximation, performance is constrained only by
  - latency: length of dependent chain of instructions
  - throughput: the number of particular types of instructions it can execute per cycle (e.g., two fmadds/cycle)
GCC is smart enough to promote an array to registers.

You can achieve the same result with a simpler code like this:

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < 10; j++) {
        x[j] = a * x[j] + y;
    }
}
```

...as long as the number of elements is *a small, compile-time constant.*

When this is the case, the compiler

- can completely unroll the inner loop
- and assign a different register to each element
What if the number of chains is a variable?

- can we have a procedure that can operate on an arbitrary number of elements?

```c
void axpy(..., long m) {
    // m : the number of elements in x
    ...
    for (i = 0; i < n; i++) {
        for (j = 0; j < m / 8; j++) {
            x[j] = a * x[j] + y;
        }
    }
}
```

- we hope it shows a similar performance (≈ peak if $m \geq 10$) and the program can handle an arbitrary number of sequences!
Yet, when we experiment . . .

<table>
<thead>
<tr>
<th>chains</th>
<th>clocks/iter</th>
<th>flops/clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12.08</td>
<td>1.33</td>
</tr>
<tr>
<td>2</td>
<td>12.09</td>
<td>2.65</td>
</tr>
<tr>
<td>3</td>
<td>12.06</td>
<td>3.98</td>
</tr>
<tr>
<td>4</td>
<td>12.25</td>
<td>5.23</td>
</tr>
<tr>
<td>5</td>
<td>12.38</td>
<td>6.47</td>
</tr>
<tr>
<td>11</td>
<td>15.43</td>
<td>11.42</td>
</tr>
<tr>
<td>14</td>
<td>19.43</td>
<td>11.54</td>
</tr>
<tr>
<td>17</td>
<td>18.55</td>
<td>14.66</td>
</tr>
<tr>
<td>21</td>
<td>26.48</td>
<td>12.69</td>
</tr>
<tr>
<td>25</td>
<td>26.79</td>
<td>14.94</td>
</tr>
<tr>
<td>29</td>
<td>34.59</td>
<td>13.42</td>
</tr>
<tr>
<td>33</td>
<td>34.38</td>
<td>15.37</td>
</tr>
<tr>
<td>37</td>
<td>42.39</td>
<td>13.97</td>
</tr>
<tr>
<td>41</td>
<td>42.63</td>
<td>15.39</td>
</tr>
<tr>
<td>45</td>
<td>51.01</td>
<td>14.12</td>
</tr>
<tr>
<td>48</td>
<td>55.49</td>
<td>13.84</td>
</tr>
</tbody>
</table>

The pattern is similar, but there are two differences:

- the throughput hits the plateau at \( \approx 1 \) cycles/iter
- the latency of a single update became \( \approx 12 \) cycles
Take a look at assembly

it looks like:

.L28:
1. vmovaps %ymm0, %ymm2
2. addq $32, %rax
3. vfmadd132ps -32(%rax), %ymm1, %ymm2
4. vmovaps %ymm2, -32(%rax)
5. cmpq %rcx, %rax
6. jne .L28

do you see the difference from the code we have seen before (whose latency = 5 cycles)?

.L3:
1. addq $1, %rax
2. vfmadd132ps %ymm0, %ymm2, %ymm1
3. cmpq %rax, %rdi
4. jne .L3
The reason of the latency (12 cycles)

- both stem from the fact that the code now involves load/stores
- according to Agner’s document, here are the latency of load/stores
  - vmovaps (load) : 3
  - vfmadd : 5
  - vmovaps (store) : 4
- this explains the 12 cycles \( (3 + 5 + 4 = 12) \) at \( m = 1 \)
The reason of the *throughput* \( \approx 1 \text{ cycles/iter} \)

- **what you need to know:**
  - *All instructions have their own throughput limits due to execution resources (dispatch ports and execution units)*

- “two *vfadd* per cycle” is just an example of it

- other instructions have their own limits. e.g.,
  - store operations = 1 per cycle
  - load operations = 2 per cycle
Dispatch ports

- each instruction (μ-operation) is dispatched to a specific *port*
  - `fmadd` → port 0 or 1
  - `load` → port 2 or 3
  - `store` → port 4
  - `int` → port 0, 1, 5, or 6
  - etc.

- each port can take only a single operation per cycle
  - this is *throughput*

- by knowing designated ports of instructions, one can reason about the throughput limit of a given loop fairly accurately

source: http://www.realworldtech.com/haswell-cpu/
Intel Architecture Code Analyzer

- a great tool to analyze throughput (and latency to some extent) limit

```
<p>| Num Of | 0 - DV | Ports pressure in cycles | | | | | |</p>
<table>
<thead>
<tr>
<th>Uops</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.2</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>0.8</td>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.2</td>
<td>0.2</td>
<td>1.0</td>
<td>0.5</td>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>CP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0*</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Total Num Of Uops: 34
So how to overcome the throughput limit?

- we obviously need to quit loading/storing data for every single fmadd

```c
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        x[j] = a * x[j] + c;
    }
}
```

- the minimum “unit” of a computation should look like:

```plaintext
1 load x[j] to a register;
2 do “a * x + c” several times on the register;
3 store the result to x[j];
```

- and run multiple independent units
Several ways to arrange computation

- take a variable at a time and run it until the end (suffer from latency)

- advance all variables one step at a time (suffer from the store throughput)

strategy 1: take a few variables and run them until the end

strategy 2: advance all variables, a few steps at a time
Implementing strategy 1

- say we advance ten elements at a time

```c
for (j = 0; j < m; j += 10) {
    /* run 10 variables until the end */
    for (i = 0; i < n; i++) {
        for (jj = j; jj < j + 10; jj++) {
            xx[jj] = a * xx[jj] + c;
        }
    }
}
```

- we hope it loads/stores each variable only once through the \(i\) loop (line 2)!
- this coding *depends on the compiler’s smartness* we have witnessed
  - promote fixed-sized arrays into registers
Implementing strategy 2

- say we advance all variables three steps at a time

```c
for (i = 0; i < n; i += 3) {
    /* run all variables 3 steps */
    for (j = 0; j < m; j++) {
        x[j] = a * x[j] + c;
        x[j] = a * x[j] + c;
        x[j] = a * x[j] + c;
    }
}
```

- again, we hope the compiler’s smartness to eliminate intermediate load/stores (purple parts)

- the latency of a single $j$ iteration increases, but we hope the $j$ loop exposes lots of independent computations
Results

- strategy 1: succeeded
  - $\approx 31.92$ flops/clock out of $32$ flops/clock; $99.75\%$ of the peak
- strategy 2: not as successful as strategy 1
  - $\approx 27.82$ flops/clock; $(87\%$ of the peak)
Reason of suboptimal results for strategy 2

- again, latency is the issue, but in a subtler way
Reason of suboptimal results for strategy 2

- again, latency is the issue, but in a subtler way
- the processor decodes instructions in the order issued by the program (*program order*)
Reason of suboptimal results for strategy 2

- again, latency is the issue, but in a subtler way
- the processor decodes instructions in the order issued by the program (program order)
- it can execute instructions (μ-operations) whose operands arrive (out of order execution)
Reason of suboptimal results for strategy 2

- again, latency is the issue, but in a subtler way
- the processor decodes instructions in the order issued by the program (*program order*)
- it can execute instructions (μ-operations) whose operands arrive (*out of order execution*)
- ⇒ operations whose operands are not ready are put in a buffer *reservation station*
Reason of suboptimal results for strategy 2

- but the reservation station (any physical resource in a processor, for that matter) is finite (Haswell has 60 entries)
Reason of suboptimal results for strategy 2

- but the reservation station (any physical resource in a processor, for that matter) is finite (Haswell has 60 entries)
- $\Rightarrow$ the processor can keep only so many operations waiting
but the reservation station (any physical resource in a processor, for that matter) is finite (Haswell has 60 entries)
⇒ the processor can keep only so many operations waiting
when the reservation station is full, the processor cannot decode any more instructions (even if some of the following instructions can be immediately executed)
Implication of finite reservation station

- instructions must enter the reservation station in the program order
- instructions near the tail of a long dependency chain occupies entries for a long time

\[ t \]

the issue order

the earliest cycle each instruction can be executed

\[ \Delta \text{ load} \quad \circ \text{ fmadd} \quad \square \text{ store} \]

- if there are too many such instructions, the processor cannot decode instructions further ahead and fail to find otherwise executable instructions
Developing near peak FLOPS matrix multiply

- let’s develop a matrix multiply that runs near peak FLOPS on Haswell

\[ C += A \times B \]

- it is a great application of the concept you have just learned
A few simplifying assumptions

- we add assumptions that $M$, $N$, and $K$ are multiple of certain numbers along the way, (forget how to process “left-over” rows/columns in this slide)
- we assume matrices are of “convenient” sizes (assume they are small to fit in a level of cache; assume they are large enough to fill pipeline)
Step 1: Baseline code

```c
for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
    for (k = 0; k < K; k++)
      C[i,j] += A[i,k] * B[k,j];
```
Step 2: Vectorization

```java
for (i = 0; i < M; i++)
    for (j = 0; j < N; j += 8)
        for (k = 0; k < K; k++)
            C[i,j:j+8] += A[i,k] * B[k,j:j+8];
```

- Assume $N$ is a multiple of SIMD lanes (8)
Step 3: Concurrently update many elements of $C$

```cpp
for (i = 0; i < M; i += 10) 
  for (j = 0; j < N; j += 8) 
    for (k = 0; k < K; k++)
      for (di = 0; di < 10; di++)
```

- Assume $M$ is a multiple of the number of elements to update concurrently (10)
- The above code stores to $C$ in every iteration; they need to be (manually) hoisted out of the loop
Step 4: Reuse an element of $A$

for (i = 0; i < M; i += 5)
    for (j = 0; j < N; j += 16)
        for (k = 0; k < K; k++)
            for (di = 0; di < 5; di++)
                for (dj = 0; dj < 16; dj += 8)

remarks: assume

- $M$ is a multiple of half the number of elements to update concurrently (5)
- $N$ is a multiple of twice the number of SIMD lanes (16)
Takeaways (1)

- Haswell peak FLOPS = “execute two \texttt{fmadds} every cycle” (no other combinations)
  - other processors have different limits, but the basic is the same
- no, single-core performance is not about reducing the number of instructions
- it’s about how to increase parallelism
  - SIMD
  - IPC
Takeaways (2)

- Dependent instructions incur latencies and hinder parallelism.

- Independent instructions are executed in parallel, up to throughput limits.

- Throughput limits are determined by dispatch ports.

- Waiting instructions occupy a reservation station entry, so you’d better avoid issuing deeply dependent instructions too early.