Parallelism and Performance

Instructor: Steven Ho
Review of Last Lecture

• Cache Performance
  – AMAT = HT + MR \times MP
Multilevel Cache Diagram

Legend:
- Request for data
- Return of data

CPU

Memory Access

L1$

Write Miss

Hit

Store

Return

Path of data back to CPU

L2$

Main Memory

Write Miss

Hit

If Write Allocate

Store
Review of Last Lecture

• Cache Performance
  – AMAT = HT + MR × MP
  – AMAT = HT₁ + MR₁ × (HT₂ + MR₂ × MP₂)

• MR_{local}: relative to number of accesses to *that* level cache

• MR_{global}: relative to total accesses to L1$!
  – MR_{global} = product of all MR_i
Review -- 3 C’s of Caches

• Compulsory
  – Never before requested that memory address
  – Can be reduced by increasing block size (spatial locality)

• Conflict
  – Due to insufficient associativity
  – check access pattern again with fully associative LRU cache with same block size

• Capacity
  – Can only be reduced by increasing cache size
  – all leftover accesses
Anatomy of a Cache Question

- Cache questions come in a few flavors:
  1) TIO Breakdown
  2) For fixed cache parameters, analyze the performance of the given code/sequence
  3) For fixed cache parameters, find best/worst case scenarios
  4) For given code/sequence, how does changing your cache parameters affect performance?
  5) AMAT
Stride vs Stretch

**Stride**
- distance between consecutive memory accesses
- compare to block size

**Stretch**
- distance between first and last accesses
- compare to cache size

7/18/2018  CS61C Su18 - Lecture 17
Question:
What is the hit rate of the second loop?

• 1 KiB direct-mapped cache with 16 B blocks

#define SIZE 2048 // 2^11

int foo() {
    int a[SIZE];
    int sum = 0;
    for(int i=0; i<SIZE; i+=256) sum += a[i];
    for(int i=0; i<SIZE; i+=256) sum += a[i];
    return sum;
}

0% (A)
25% (B)
50% (C)
100% (D)

Step size:
256 integers = 1024 B = 1 KiB

1 KiB Cache
Question: What is the hit rate of the second loop?

- 1 KiB fully-associative cache with 16 B blocks

```c
#define SIZE 2048 // 2^11
int foo() {
    int a[SIZE];
    int sum = 0;
    for(int i=0; i<SIZE; i+=256) sum += a[i];
    for(int i=0; i<SIZE; i+=256) sum += a[i];
    return sum;
}
```

(A) 0%  (B) 25%  (C) 50%  (D) 100%
Agenda

• Performance
• Administrivia
• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Meet the Staff
• Intel SSE Intrinsics
Great Idea #4: Parallelism

**Software**

- Parallel Requests
  Assigned to computer
  e.g. search “Garcia”

- Parallel Threads
  Assigned to core
  e.g. lookup, ads

- Parallel Instructions
  > 1 instruction @ one time
  e.g. 5 pipelined instructions

- Parallel Data
  > 1 data item @ one time
  e.g. add of 4 pairs of words

**Hardware**

- Warehouse Scale Computer

Leverage Parallelism & Achieve High Performance

We were here

We were here

Software        Hardware

Cache Memory

Core

Input/Output

Instruction Unit(s)

Functional Unit(s)

Logic Gates

Smart Phone

7/18/2018

CS61C Su18 - Lecture 17
Measurements of Performance

• *Latency* (or *response time* or *execution time*)
  – Time to complete one task

• *Bandwidth* (or *throughput*)
  – Tasks completed per unit time
Defining CPU Performance

- What does it mean to say X is faster than Y?

**Tesla vs. School Bus:**
- 2015 Tesla Model S P90D (Ludicrous Speed Upgrade)
  - 5 passengers, 2.8 secs in quarter mile
- 2011 Type D school bus
  - Up to 90 passengers, quarter mile time?
Cloud Performance: Why Application Latency Matters

- Key figure of merit: application responsiveness
  - Longer the delay, the fewer the user clicks, the less the user happiness, and the lower the revenue per user

<table>
<thead>
<tr>
<th>Server Delay (ms)</th>
<th>Increased time to next click (ms)</th>
<th>Queries/user</th>
<th>Any clicks/user</th>
<th>User satisfaction</th>
<th>Revenue/User</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>200</td>
<td>500</td>
<td>--</td>
<td>-0.3%</td>
<td>-0.4%</td>
<td>--</td>
</tr>
<tr>
<td>500</td>
<td>1200</td>
<td>--</td>
<td>-1.0%</td>
<td>-0.9%</td>
<td>-1.2%</td>
</tr>
<tr>
<td>1000</td>
<td>1900</td>
<td>-0.7%</td>
<td>-1.9%</td>
<td>-1.6%</td>
<td>-2.8%</td>
</tr>
<tr>
<td>2000</td>
<td>3100</td>
<td>-1.8%</td>
<td>-4.4%</td>
<td>-3.8%</td>
<td>-4.3%</td>
</tr>
</tbody>
</table>

Figure 6.10 Negative impact of delays at Bing search server on user behavior [Brutlag and Schurman 2009].
Defining Relative Performance

• Compare performance (perf) of X vs. Y
  – Latency in this case

• \( \text{Perf}_X = \frac{1}{\text{Program Execution Time}_X} \)

• \( \text{Perf}_X > \text{Perf}_Y \)
  \[ \rightarrow \quad \text{Execution Time}_Y > \text{Execution Time}_X \]

• “Computer X is N times faster than Y”:

\[
\frac{\text{Performance}_X}{\text{Performance}_Y} = \frac{\text{Execution Time}_Y}{\text{Execution Time}_X} = N
\]
Measuring CPU Performance

• Computers use a clock to determine when events take place within hardware
• *Clock cycles*: discrete time intervals
  – a.k.a. clocks, cycles, clock periods, clock ticks
• *Clock rate* or *clock frequency*: clock cycles per second (inverse of clock cycle time)

• Example: 3 GigaHertz clock rate means clock cycle time = 1/(3x10^9) seconds
  = 333 picoseconds (ps)
CPU Performance Factors

• To distinguish between processor time and I/O, *CPU time* is time spent in processor

\[
\frac{\text{CPU Time}}{\text{Program}} = \frac{\text{Clock Cycles}}{\text{Program}} \times \text{Clock Cycle Time}
\]

\[
= \frac{\text{Clock Cycles}}{\text{Program}} \times \frac{1}{\text{Clock Rate}}
\]
CPU Performance Factors

• But a program executes instructions!
  – Let’s reformulate this equation, then:

\[
\frac{\text{CPU Time}}{\text{Program}} = \frac{\text{Clock Cycles}}{\text{Program}} \times \text{Clock Cycle Time}
\]

\[
= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Average Clock Cycles}}{\text{Instruction}} \times \frac{1}{\text{Clock Rate}}
\]

- Instruction Count
- CPI: Clock Cycles Per Instruction
- Clock Cycle Time
CPU Performance Equation

• For a given program:

CPU Time

\[ \text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time} \]

\[ = \text{Instructions} \times \text{CPI} \times \frac{1}{\text{Clock Rate}} \]

– The “Iron Law” of processor performance
Question: Which statement is TRUE?

- Computer A clock cycle time 250 ps, $\text{CPI}_A = 2$
- Computer B clock cycle time 500 ps, $\text{CPI}_B = 1.2$
- Assume A and B have the same instruction set

\[
\text{CPU Time} = \text{Instructions} \times \text{CPI} \times \text{Clock Cycle Time}
\]

(A) Computer A is $\approx 1.2$ times faster than B
(B) Computer A is $\approx 4.0$ times faster than B
(C) Computer B is $\approx 1.7$ times faster than A
(D) Computer B is $\approx 3.4$ times faster than A
Workload and Benchmark

- **Workload**: Set of programs run on a computer
  - Actual collection of applications run or made from real programs to approximate such a mix
  - Specifies programs, inputs, and relative frequencies

- **Benchmark**: Program selected for use in comparing computer performance
  - Benchmarks form a workload
  - Usually standardized so that many use them
SPEC
(System Performance Evaluation Cooperative)

• Computer vendor cooperative for benchmarks, started in 1989

• SPECCPU2006
  – 12 Integer Programs
  – 17 Floating-Point Programs

• SPECratio: reference execution time on old reference computer divide by execution time on new computer to get an effective speed-up
  – Want number where bigger is faster
## SPECINT2006 on AMD Barcelona

<table>
<thead>
<tr>
<th>Description</th>
<th>Instruction Count (B)</th>
<th>CPI</th>
<th>Clock cycle time (ps)</th>
<th>Execution Time (s)</th>
<th>Reference Time (s)</th>
<th>SPEC-ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreted string processing</td>
<td>2,118</td>
<td>0.75</td>
<td>400</td>
<td>637</td>
<td>9,770</td>
<td>15.3</td>
</tr>
<tr>
<td>Block-sorting compression</td>
<td>2,389</td>
<td>0.85</td>
<td>400</td>
<td>817</td>
<td>9,650</td>
<td>11.8</td>
</tr>
<tr>
<td>GNU C compiler</td>
<td>1,050</td>
<td>1.72</td>
<td>400</td>
<td>724</td>
<td>8,050</td>
<td>11.1</td>
</tr>
<tr>
<td>Combinatorial optimization</td>
<td>336</td>
<td>10.0</td>
<td>400</td>
<td>1,345</td>
<td>9,120</td>
<td>6.8</td>
</tr>
<tr>
<td>Go game</td>
<td>1,658</td>
<td>1.09</td>
<td>400</td>
<td>721</td>
<td>10,490</td>
<td>14.6</td>
</tr>
<tr>
<td>Search gene sequence</td>
<td>2,783</td>
<td>0.80</td>
<td>400</td>
<td>890</td>
<td>9,330</td>
<td>10.5</td>
</tr>
<tr>
<td>Chess game</td>
<td>2,176</td>
<td>0.96</td>
<td>400</td>
<td>837</td>
<td>12,100</td>
<td>14.5</td>
</tr>
<tr>
<td>Quantum computer simulation</td>
<td>1,623</td>
<td>1.61</td>
<td>400</td>
<td>1,047</td>
<td>20,720</td>
<td>19.8</td>
</tr>
<tr>
<td>Video compression</td>
<td>3,102</td>
<td>0.80</td>
<td>400</td>
<td>993</td>
<td>22,130</td>
<td>22.3</td>
</tr>
<tr>
<td>Discrete event simulation library</td>
<td>587</td>
<td>2.94</td>
<td>400</td>
<td>690</td>
<td>6,250</td>
<td>9.1</td>
</tr>
<tr>
<td>Games/path finding</td>
<td>1,082</td>
<td>1.79</td>
<td>400</td>
<td>773</td>
<td>7,020</td>
<td>9.1</td>
</tr>
<tr>
<td>XML parsing</td>
<td>1,058</td>
<td>2.70</td>
<td>400</td>
<td>1,143</td>
<td>6,900</td>
<td>6.0</td>
</tr>
</tbody>
</table>
### Which System is Faster?

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>

a) System A  
b) System B  
c) Same performance  
d) Unanswerable question
... Depends on Who’s Selling

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>10</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>B</td>
<td>20</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

Average throughput

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.50</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>B</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

Throughput relative to B

<table>
<thead>
<tr>
<th>System</th>
<th>Rate (Task 1)</th>
<th>Rate (Task 2)</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>B</td>
<td>2.00</td>
<td>0.50</td>
<td>1.25</td>
</tr>
</tbody>
</table>

Throughput relative to A
Agenda

• Performance
• Administrivia
• Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Meet the Staff
• Intel SSE Intrinsics
Adminstrivia

• Lab on Thurs. is for catching up
• HW5 due 7/23, Proj3 due 7/20
• Proj 3 party on Fri (7/20), 4-6PM @Woz
• Guerilla Session on Wed. 4-6pm @Soda 405
• “Lost” Discussion Sat. Cory 540AB, 12-2PM
• Midterm 2 is coming up! Next Wed. in lecture
  – Covering up to Performance
  – Review Session Sunday 2-4pm @GPB 100
  – There will be discussion after MT2 :(
Agenda

• Performance
• Administrivia
• Parallelism and Flynn’s Taxonomy
• Data Level Parallelism and SIMD
• Meet the Staff
• Intel SSE Intrinsics
Hardware vs. Software Parallelism

- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn’s Taxonomy* is for parallel hardware
Flynn’s Taxonomy

- SIMD and MIMD most commonly encountered today
- Most common parallel programming style:
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, audio/video

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Single</th>
<th>Multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>SISD: Intel Pentium 4</td>
<td>SIMD: SSE instructions of x86</td>
</tr>
<tr>
<td>Multiple</td>
<td>MISD: No examples today</td>
<td>MIMD: Intel Xeon e5345 (Clovertown)</td>
</tr>
</tbody>
</table>
Single Instruction/Single Data Stream

- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines
Multiple Instruction/Single Data Stream

- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)

- MISD no longer commonly encountered, mainly of historical interest only
Single Instruction/Multiple Data Stream

- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized (e.g. SIMD instruction extensions or Graphics Processing Unit)
Multiple Instruction/Multiple Data Stream

- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers
Agenda

• Performance
• Administrivia
• Parallelism and Flynn’s Taxonomy
  • Data Level Parallelism and SIMD
• Meet the Staff
• Intel SSE Intrinsics
Great Idea #4: Parallelism

**Software**

- **Parallel Requests**
  Assigned to computer
  e.g. search “Garcia”

- **Parallel Threads**
  Assigned to core
  e.g. lookup, ads

- **Parallel Instructions**
  > 1 instruction @ one time
  e.g. 5 pipelined instructions

- **Parallel Data**
  > 1 data item @ one time
  e.g. add of 4 pairs of words

- **Hardware descriptions**
  All gates functioning in parallel at same time

**Hardware**

- Warehouse
- Scale
- Computer

_Leverage Parallelism & Achieve High Performance_

**Computer**

- Core
- …
- Core
- Memory
- Input/Output

**Instruction Unit(s)**

- Functional Unit(s)
  - $A_0 + B_0$
  - $A_1 + B_1$
  - $A_2 + B_2$
  - $A_3 + B_3$

**Cache Memory**

**Logic Gates**

**Smart Phone**

**Warehouse**

**Leverage Parallelism & Achieve High Performance**

**Computer**

**Core**

**Input/Output**

**Smart Phone**

**We are here**

**Cache Memory**

**Logic Gates**

**Smart Phone**

**Kernel**

**Parallel Data**

> 1 data item @ one time

**Parallel Instructions**

> 1 instruction @ one time

**Parallel Requests**

Assigned to computer

**Parallel Threads**

Assigned to core

**Parallel Data**

We are here

**Hardware descriptions**

All gates functioning in parallel at same time

**Core**

**Input/Output**

**Logic Gates**

**Kernel**

**Parallel Requests**

Assigned to computer

e.g. search “Garcia”

**Parallel Threads**

Assigned to core

e.g. lookup, ads

**Parallel Instructions**

> 1 instruction @ one time

e.g. 5 pipelined instructions

**Parallel Data**

> 1 data item @ one time

e.g. add of 4 pairs of words

**Hardware descriptions**

All gates functioning in parallel at same time
SIMD Architectures

• *Data-Level Parallelism (DLP):* Executing one operation on multiple data streams

• **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

\[ y[i] := c[i] \times x[i], \ 0 \leq i < n \]

• **Sources of performance improvement:**
  – One instruction is fetched & decoded for entire operation
  – Multiplications are known to be independent
  – Pipelining/concurrency in memory access as well
Example: SIMD Array Processing

for each f in array
    f = sqrt(f)

for each f in array {
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}

for each 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
“Advanced Digital Media Boost”

• To improve performance, Intel’s SIMD instructions
  – Fetch one instruction, do the work of multiple instructions
  – MMX (MultiMedia eXtension, Pentium II processor family)
  – SSE (Streaming SIMD Extension, Pentium III and beyond)
SSE Instruction Categories for Multimedia Support

<table>
<thead>
<tr>
<th>Instruction Category</th>
<th>Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Saturating add/subtract</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Max/min/minimum</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Average</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
<tr>
<td>Shift right/left</td>
<td>Eight 8-bit or Four 16-bit</td>
</tr>
</tbody>
</table>

- Intel processors are **CISC (complicated instrs)**
- SSE-2+ supports wider data types to allow $16 \times 8$-bit and $8 \times 16$-bit operands
In Intel Architecture (unlike RISC-V) a word is 16 bits

- Single precision FP: Double word (32 bits)
- Double precision FP: Quad word (64 bits)
XMM Registers

- Architecture extended with eight 128-bit data registers
  - 64-bit address architecture: available as 16 64-bit registers (XMM8 – XMM15)
  - e.g. 128-bit packed single-precision floating-point data type (doublewords), allows four single-precision operations to be performed simultaneously

<table>
<thead>
<tr>
<th>127</th>
<th>XMM7</th>
</tr>
</thead>
<tbody>
<tr>
<td>126</td>
<td>XMM6</td>
</tr>
<tr>
<td>125</td>
<td>XMM5</td>
</tr>
<tr>
<td>124</td>
<td>XMM4</td>
</tr>
<tr>
<td>123</td>
<td>XMM3</td>
</tr>
<tr>
<td>122</td>
<td>XMM2</td>
</tr>
<tr>
<td>121</td>
<td>XMM1</td>
</tr>
<tr>
<td>120</td>
<td>XMM0</td>
</tr>
</tbody>
</table>
## SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV{A/U}{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD{SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP{SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MOVE{H/L}{PS/PD} xmm, mem/xmm</td>
<td>MUL{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV{SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQR{T}{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN{SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

{SS} Scalar Single precision FP: 1 32-bit operand in a 128-bit register

{PS} Packed Single precision FP: 4 32-bit operands in a 128-bit register

{SD} Scalar Double precision FP: 1 64-bit operand in a 128-bit register

{PD} Packed Double precision FP, or 2 64-bit operands in a 128-bit register
### SSE/SSE2 Floating Point Instructions

<table>
<thead>
<tr>
<th>Data transfer</th>
<th>Arithmetic</th>
<th>Compare</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV {A/U} {SS/PS/SD/PD} xmm, mem/xmm</td>
<td>ADD {SS/PS/SD/PD} xmm, mem/xmm</td>
<td>CMP {SS/PS/SD/PD}</td>
</tr>
<tr>
<td></td>
<td>SUB {SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td>MOV {H/L} {PS/PD} xmm, mem/xmm</td>
<td>MUL {SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DIV {SS/PS/SD/PD} xmm, mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SQRT {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MAX {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIN {SS/PS/SD/PD} mem/xmm</td>
<td></td>
</tr>
</tbody>
</table>

**xmm**: one operand is a 128-bit SSE2 register  
**mem/xmm**: other operand is in memory or an SSE2 register  

- **{A}** 128-bit operand is aligned in memory  
- **{U}** means the 128-bit operand is unaligned in memory  
- **{H}** means move the high half of the 128-bit operand  
- **{L}** means move the low half of the 128-bit operand
Example: Add Single Precision FP Vectors

Computation to be performed:

\[
\begin{align*}
\text{vec\_res.} \text{x} &= \text{v1.} \text{x} + \text{v2.} \text{x}; \\
\text{vec\_res.} \text{y} &= \text{v1.} \text{y} + \text{v2.} \text{y}; \\
\text{vec\_res.} \text{z} &= \text{v1.} \text{z} + \text{v2.} \text{z}; \\
\text{vec\_res.} \text{w} &= \text{v1.} \text{w} + \text{v2.} \text{w};
\end{align*}
\]

SSE Instruction Sequence:

\[
\begin{align*}
\text{movaps} & \text{ address-of-v1, } \text{%xmm0} \\
& \text{ // v1.w} | \text{v1.z} | \text{v1.y} | \text{v1.x} \rightarrow \text{xmm0} \\
\text{addps} & \text{ address-of-v2, } \text{%xmm0} \\
& \text{ // v1.w+v2.w} | \text{v1.z+v2.z} | \text{v1.y+v2.y} | \text{v1.x+v2.x} \rightarrow \text{xmm0} \\
\text{movaps} & \text{ %xmm0, address-of-vec\_res}
\end{align*}
\]

\text{move} from mem to XMM register, memory aligned, packed single precision

\text{add} from mem to XMM register, packed single precision

\text{move} from XMM register to mem, memory aligned, packed single precision