

Лекция 4

Векторизация кода (code vectorization: SSE/AVX)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Высокопроизводительные вычислительные системы»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

Instruction Level Parallelism

Архитектурные решения для обеспечения параллельного выполнения коротких последовательностей инструкций

- **Вычислительный конвейер (Pipeline)** – совмещение (overlap) во времени выполнения инструкций
- **Суперскалярное выполнение (Superscalar)** – выдача и выполнение нескольких инструкций за такт ($CPI < 1$)
- **Внеочередное выполнение (Out-of-order execution)** – динамическая выдача инструкций на выполнение по готовности их данных

Instruction Level Parallelism

Архитектурные решения для обеспечения параллельного выполнения коротких последовательностей инструкций

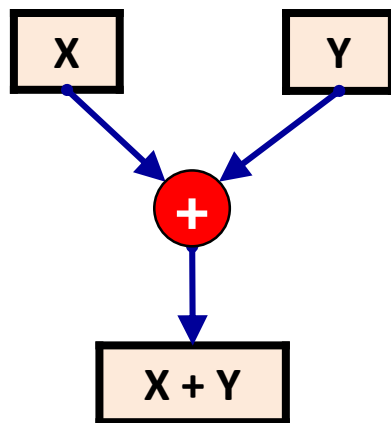
- **Вычислительный конвейер (Pipeline)** – совмещение (overlap) во времени выполнения инструкций
- **Суперскалярное выполнение (Superscalar)** – выдача и выполнение нескольких инструкций за такт ($CPI < 1$)
- **Внеочередное выполнение (Out-of-order execution)** – динамическая выдача инструкций на выполнение по готовности их данных

**Каждый подход имеет свои плюсы и минусы
Существует ли альтернатива этим подходам?**

Векторные процессоры

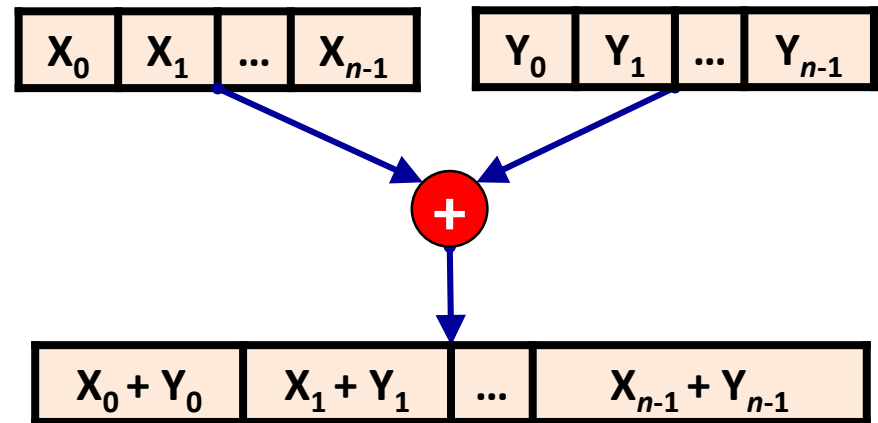
- Векторный процессор (Vector processor) – это процессор, поддерживающий на уровне системы команд операции для работы с одномерными массивами – векторами (vector)

Скалярный процессор
(Scalar processor)



`add Z, X, Y`

Векторный процессор
(Vector processor)



`add.v Z[0:n-1], X[0:n-1], Y[0:n-1]`

Vector CPU vs. Scalar CPU

Поэлементное суммирование двух массивов из 10 чисел

Скалярный процессор (Scalar processor)

```
for i = 1 to 10 do
  IF - Instruction Fetch (next)
  ID - Instruction Decode
  Load Operand1
  Load Operand2
  Add Operand1 Operand2
  Store Result
end for
```

Векторный процессор (Vector processor)

```
IF - Instruction Fetch
ID - Instruction Decode
Load Operand1[0:9]
Load Operand2[0:9]
Add Operand1[0:9] Operand2[0:9]
Store Result
```

Vector CPU vs. Scalar CPU

Поэлементное суммирование двух массивов из 10 чисел

Скалярный процессор (Scalar processor)

```
for i = 1 to 10 do
  IF - Instruction Fetch (next)
  ID - Instruction Decode
  Load Operand1
  Load Operand2
  Add Operand1 Operand2
  Store Result
end for
```

Векторный процессор (Vector processor)

```
IF - Instruction Fetch
ID - Instruction Decode
Load Operand1[0:9]
Load Operand2[0:9]
Add Operand1[0:9] Operand2[0:9]
Store Result
```

- Меньше преобразований адресов
- Меньше IF, ID
- Меньше конфликтов конвейера, ошибок предсказания переходов
- Эффективнее доступ к памяти (2 выборки vs. 20)
- Операция над операндами выполняется параллельно
- Уменьшился размер кода

Производительность векторного процессора

Факторы влияющие на производительность векторного процессора

- Доля кода, который может быть выражен в векторной форме (с использованием векторных инструкций)
- Длина вектора (векторного регистра)
- Латентность векторной инструкции (Vector startup latency) – начальная задержка конвейера при обработке векторной инструкции
- Количество векторных регистров
- Количество векторных модулей доступа к памяти (load-store)
- ...

Виды векторных процессоров

- **Векторные процессоры память-память**
(Memory-memory vector processor) – векторы размещены в оперативной памяти, все векторные операции память-память
- Примеры:
 - ❑ CDC STAR-100 (1972, вектор 65535 элементов)
 - ❑ Texas Instruments ASC (1973)
- **Векторные процессоры регистр-регистр**
(Register-vector vector processor) – векторы размещены в векторных регистрах, все векторные операции выполняются между векторными регистрами
- Примеры: практически все векторные системы начиная с конца 1980-х: Cray, Convex, Fujitsu, Hitachi, NEC, ...

Векторные вычислительные системы

- Cray 1 (1976) 80 MHz, 8 regs, 64 elems
- Cray XMP (1983) 120 MHz 8 regs, 64 elems
- Cray YMP (1988) 166 MHz 8 regs, 64 elems
- Cray C-90 (1991) 240 MHz 8 regs, 128 elems
- Cray T-90 (1996) 455 MHz 8 regs, 128 elems
- Conv. C-1 (1984) 10 MHz 8 regs, 128 elems
- Conv. C-4 (1994) 133 MHz 16 regs, 128 elems
- Fuj. VP200 (1982) 133 MHz 8-256 regs, 32-1024 elems
- Fuj. VP300 (1996) 100 MHz 8-256 regs, 32-1024 elems
- NEC SX/2 (1984) 160 MHz 8+8K regs, 256+var elems
- NEC SX/3 (1995) 400 MHz 8+8K regs, 256+var elems

Модули векторного процессора

- **Векторные функциональные устройства**
(Vector Functional Units): полностью конвейеризированы,
FP add, FP mul, FP reciprocal ($1/x$), Int. add, Int. mul, ...
- **Векторные модули доступа к памяти**
(Vector Load-Store Units)
- **Векторные регистры** (Vector Registers):
регистры фиксированной длины (как правило, 64-512 бит)

SIMD-инструкции современных процессоров

- Intel **MMX**: 1997, Intel Pentium MMX, IA-32
- AMD **3DNow!**: 1998, AMD K6-2, IA-32
- Apple, IBM, Motorola **AltiVec**: 1998, PowerPC G4, G5, IBM Cell/POWER
- Intel **SSE** (Streaming SIMD Extensions): 1999, Intel Pentium III
- Intel **SSE2**: 2001, Intel Pentium 4, IA-32
- Intel **SSE3**: 2004, Intel Pentium 4 Prescott, IA-32
- Intel **SSE4**: 2006, Intel Core, AMD K10, x86-64
- AMD **SSE5** (XOP, FMA4, CVT16): 2007, 2009, AMD Bulldozer
- Intel **AVX**: 2008, Intel Sandy Bridge
- ARM **Advanced SIMD (NEON)**: ARMv7, ARM Cortex A
- MIPS **SIMD Architecture (MSA)**: 2012, MIPS R5
- Intel **AVX2**: 2013, Intel Haswell
- Intel **AVX-512**: 2013, Intel Xeon Skylake (2015), Intel Xeon Phi
- ...

CPUID (CPU Identification): Microsoft Windows

Windows CPU-Z

The screenshot shows the CPU-Z application window with the 'CPU' tab selected. The processor is an Intel Core i5 2520M, Sandy Bridge architecture, 32 nm technology, running at 2.50 GHz. The instructions supported are MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x, AES, and AVX. The cache configuration includes 2 x 32 KBytes L1 Data and L1 Inst., 2 x 256 KBytes Level 2, and 3 MBytes Level 3. The core speed is 797.4 MHz, multiplier is x 8.0, and bus speed is 99.7 MHz. The selection is set to Processor #1, showing 2 cores and 4 threads.

Processor			
Name	Intel Core i5 2520M		
Code Name	Sandy Bridge	Max TDP	35 W
Package	Socket 988B rPGA		
Technology	32 nm	Core VID	0.766 V
Specification	Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz		
Family	6	Model	A
Ext. Family	6	Ext. Model	2A
Stepping	7	Revision	D2
Instructions	MMX, SSE (1, 2, 3, 3S, 4.1, 4.2), EM64T, VT-x, AES, AVX		

Clocks (Core #0)		
Core Speed	797.4 MHz	
Multiplier	x 8.0	
Bus Speed	99.7 MHz	
Rated FSB		

Cache		
L1 Data	2 x 32 KBytes	8-way
L1 Inst.	2 x 32 KBytes	8-way
Level 2	2 x 256 KBytes	8-way
Level 3	3 MBytes	12-way

Selection: Processor #1 | Cores: 2 | Threads: 4

CPU-Z Version 1.58 | Validate | OK

- MMX
- SSE
- AVX
- AES
- ...

CPUID (CPU Identification): GNU/Linux

- Файл `/proc/cpuinfo`: в поле `flags` хранится информация о процессоре
- Файл `/sys/devices/system/cpu/cpuX/microcode/processor_flags`
- Устройство `/dev/cpu/CPUNUM/cpuid`: чтение выполняется через `lseek` и `pread` (требуется загрузка модуля ядра `cpuid`)

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id     : GenuineIntel
cpu family    : 6
model name    : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
...
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep
mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2
ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon
pebs bts nopl xtopology nonstop_tsc aperfmperf pni pclmulqdq
dtes64 ds_cpl vmx smx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1
sse4_2 x2apic popcnt tsc_deadline_timer xsave avx lahf_lm ida
arat epb xsaveopt pln pts dtherm tpr_shadow vnmi flexpriority
ept vpid
```

CPUID (CPU Identification): Microsoft Windows

```
#include <intrin.h>

int isAVXSupported()
{
    bool AVXSupported = false;

    int cpuInfo[4];
    __cpuid(cpuInfo, 1);

    bool osUsesXSAVE_XRSTORE = cpuInfo[2] & (1 << 27) || false;
    bool cpuAVXSupport = cpuInfo[2] & (1 << 28) || false;

    if (osUsesXSAVE_XRSTORE && cpuAVXSupport) {
        // Check if the OS will save the YMM registers
        unsigned long long xcrFeatureMask =
            _xgetbv(_XCR_XFEATURE_ENABLED_MASK);
        AVXSupported = (xcrFeatureMask & 0x6) || false;
    }
    return AVXSupported;
}
```

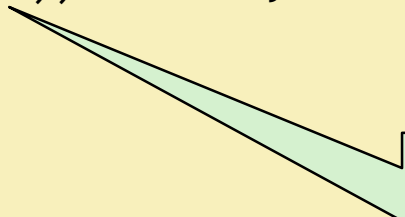
CPUID (CPU Identification): GNU/Linux

```
inline void cpuid(int fn, unsigned int *eax, unsigned int *ebx,
                  unsigned int *ecx, unsigned int *edx)
{
    asm volatile("cpuid"
                 : "=a" (*eax), "=b" (*ebx), "=c" (*ecx), "=d" (*edx)
                 : "a" (fn));
}

int is_avx_supported()
{
    unsigned int eax, ebx, ecx, edx;

    cpuid(1, &eax, &ebx, &ecx, &edx);
    return (ecx & (1 << 28)) ? 1 : 0;
}

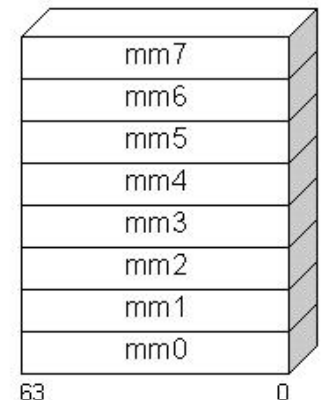
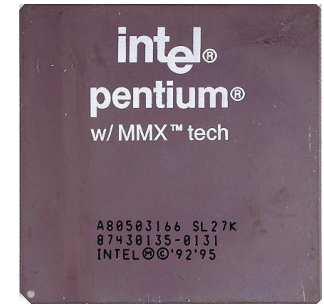
int main()
{
    printf("AVX supported: %d\n",
           is_avx_supported());
    return 0;
}
```



Intel 64 and IA-32 Architectures
Software Developer's Manual
(Vol. 2A)

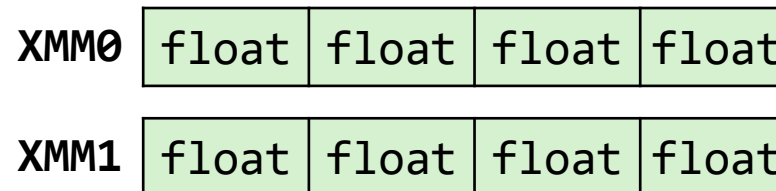
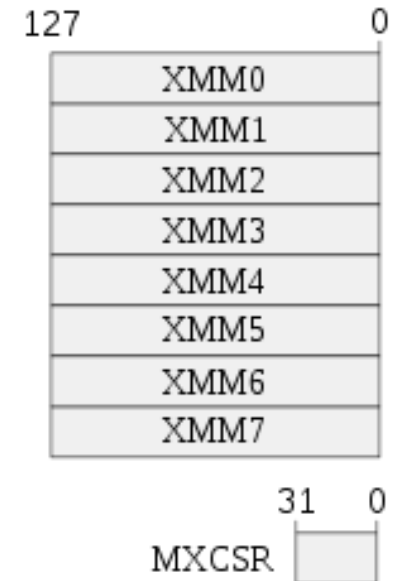
Intel MMX

- 1997, Intel Pentium MMX
- MMX – набор SIMD-инструкции обработки целочисленных векторов длиной 64 бит
- 8 виртуальных регистров mm0, mm1, ..., mm7 – ссылки на физические регистры x87 FPU (ОС не требуется сохранять/восстанавливать регистры mm0, ..., mm7 при переключении контекста)
- Типы векторов: 8 x 1 char, 4 x short int, 2 x int
- MMX-инструкции разделяли x87 FPU с FP-инструкциями – требовалось оптимизировать поток инструкций (отдавать предпочтение инструкциям одного типа)



Intel SSE

- 1999, Pentium III
- 8 векторных регистров шириной 128 бит:
`%xmm0, %xmm1, ..., %xmm7`
- Типы данных: `float` (4 элемента на вектор)
- 70 инструкций: команды пересылки, арифметические команды, команды сравнения, преобразования типов, побитовые операции
- Инструкции явной предвыборки данных, контроля кэширования данных и контроля порядка операций сохранения



```
mulps %xmm1, %xmm0 // xmm0 = xmm0 * xmm1
```

Intel SSE

- Один из разработчиков расширения SSE – ***В.М. Пентковский (1946 – 2012 г.)***
- До переход в Intel являлся сотрудником Новосибирского филиала ИТМиВТ (программное обеспечение многопроцессорных комплексов Эльбрус 1 и 2, язык Эль-76, процессор Эль-90, ...)
- Jagannath Keshava and Vladimir Pentkovski: **Pentium III Processor Implementation Tradeoffs**. // Intel Technology Journal. — 1999. — Т. 3. — № 2.
- Srinivas K. Raman, Vladimir M. Pentkovski, Jagannath Keshava: **Implementing Streaming SIMD Extensions on the Pentium III Processor**. // IEEE Micro, Volume 20, Number 1, January/February 2000: 47-57 (2000)

Intel SSE2

- 2001, Pentium 4, IA32, x86-64 (Intel 64, 2004)
- **16** векторных регистров шириной 128 бит:
%xmm0, %xmm1, ..., %xmm7; %xmm8, ..., %xmm15
- Добавлено 144 инструкции к 70 инструкциям SSE
- По сравнению с SSE сопроцессор FPU (x87) обеспечивает более точный результат при работе с вещественными числами

16 x char	char	char	char	char	char	...	char
8 x short int	short int	short int	...	short int			
4 x float int	float	float	float	float			
2 x double	double	double					
1 x 128-bit int	128-bit integer						

Intel SSE3 & SSE4

- **Intel SSE3:** 2003, Pentium 4 Prescott, IA32, x86-64 (Intel 64, 2004)
- Добавлено 13 новых инструкции к инструкциям SSE2
- Возможность горизонтальной работы с регистрами – команды сложения и вычитания нескольких значений, хранящихся в одном регистре

- **Intel SSE4:** 2006, Intel Core, AMD Bulldozer
- Добавлено 54 новых инструкции:
 - SSE 4.1: 47 инструкций, Intel Penryn
 - SSE 4.2: 7 инструкций, Intel Nehalem

Horizontal instruction

XMM0	a3	a2	a1	a0
------	----	----	----	----

XMM1	b3	b2	b1	b0
------	----	----	----	----

haddps %xmm1, %xmm0

XMM1	b3+b2	b1+b0	a3+a2	a1+a0
------	-------	-------	-------	-------

Intel AVX

- 2008, Intel Sandy Bridge (2011), AMD Bulldozer (2011)
- Размер векторов увеличен до 256 бит
- Векторные регистры переименованы: ymm0, ymm1, ..., ymm15
- Регистры xmm# – это младшие 128 бит регистров ymm#
- Трехоперандный синтаксис AVX-инструкций: $C = A + B$
- Использование ymm регистров требует поддержки со стороны операционной системы (для сохранения регистров при переключении контекстов)
 - Linux ядра $\geq 2.6.30$
 - Apple OS X 10.6.8
 - Windows 7 SP 1
- Поддержка компиляторами:
 - GCC 4.6
 - Intel C++ Compiler 11.1
 - Microsoft Visual Studio 2010
 - Open64 4.5.1

	255	128	0
YMM0		XMM0	
YMM1		XMM1	
YMM2		XMM2	
YMM3		XMM3	
YMM4		XMM4	
YMM5		XMM5	
YMM6		XMM6	
YMM7		XMM7	
YMM8		XMM8	
YMM9		XMM9	
YMM10		XMM10	
YMM11		XMM11	
YMM12		XMM12	
YMM13		XMM13	
YMM14		XMM14	
YMM15		XMM15	

Формат SSE-инструкций

ADDPS

- Название инструкции

- Тип инструкции

S – над скаляром (scalar)

P – над упакованным вектором (packed)

- Тип элементов вектора/скаляра

S – single precision (float, 32-бита)

D – double precision (double, 64-бита)

- **ADDPS** – add 4 packed single-precision values (float)
- **ADDSD** – add 1 scalar double-precision value (double)

Скалярные SSE-инструкции

- **Скалярные SSE-инструкции** (scalar instruction) – в операции участвуют только младшие элементы данных (скаляры) в векторных регистрах/памяти
- ADDSS, SUBSS, MULSS, DIVSS, ADDSD, SUBSD, MULSD, DIVSD, SQRTSS, RSQRTSS, RCPSS, MAXSS, MINSS, ...

Scalar Single-precision (float)

XMM0	4.0	3.0	2.0	1.0
XMM1	7.0	7.0	7.0	7.0

addss %xmm0, %xmm1

XMM1	4.0	3.0	2.0	8.0
------	-----	-----	-----	-----

- Результат помещается в младшее двойное слово (32-bit) операнда-назначения (xmm1)
- Три старших двойных слова из операнда-источника (xmm0) копируются в операнд-назначение (xmm1)

Scalar Double-precision (double)

XMM0	8.0	6.0
XMM1	7.0	7.0

addsd %xmm0, %xmm1

XMM1	8.0	13.0
------	-----	------

- Результат помещается в младшие 64 бита операнда-назначения (xmm1)
- Старшие 64 бита из операнда-источника (xmm0) копируются в операнд-назначение (xmm1)

Инструкции над упакованными векторами

- **SSE-инструкция над упакованными векторами** (packed instruction) – в операции участвуют все элементы данных векторных регистров/памяти
- ADDPS, SUBPS, MULPS, DIVPS, ADDPD, SUBPD, MULPD, DIVPD, SQRTPS, RSQRTPS, RCPPS, MAXPS, MINPS, ...

Packed Single-precision (float)

XMM0	4.0	3.0	2.0	1.0
XMM1	7.0	7.0	7.0	7.0

addps %xmm0, %xmm1

XMM1	11.0	10.0	9.0	8.0
------	------	------	-----	-----

Packed Double-precision (double)

XMM0	8.0	6.0
XMM1	7.0	7.0

addpd %xmm0, %xmm1

XMM1	15.0	13.0
------	------	------

Инструкций SSE

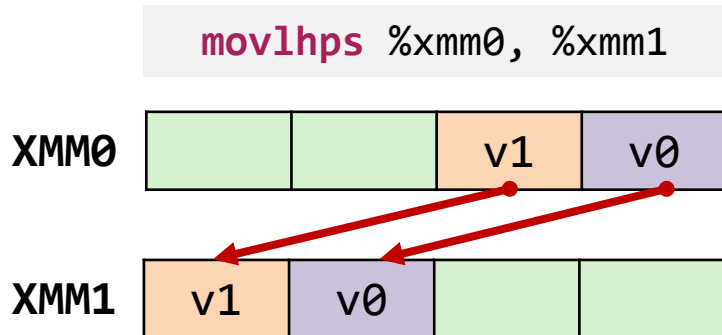
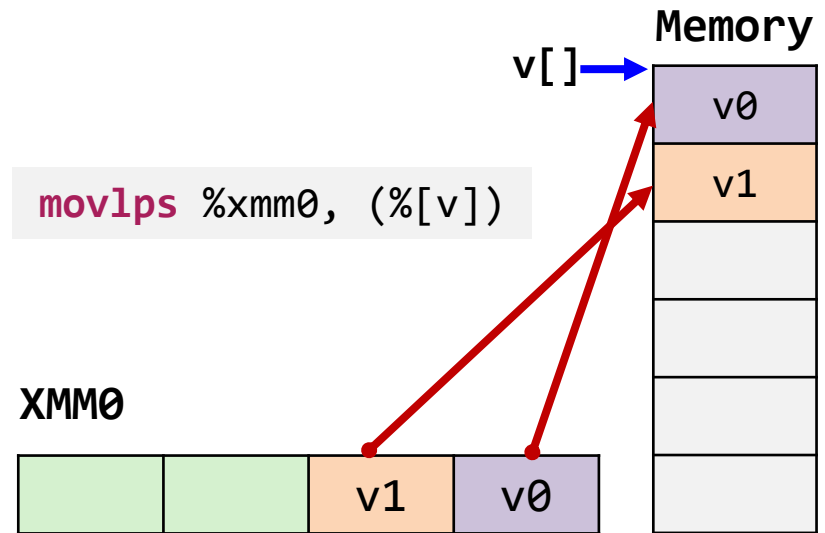
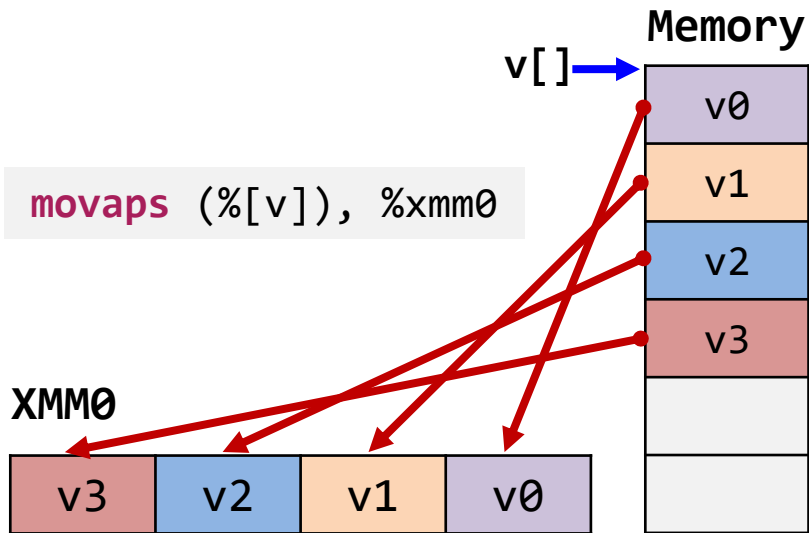
- **Операции копирования данных (mem-reg/reg-mem/reg-reg)**
 - Scalar: MOVSS
 - Packed: MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS
- **Арифметические операции**
 - Scalar: ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS
 - Packed: ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
- **Операции сравнения**
 - Scalar: CMPSS, COMISS, UCOMISS
 - Packed: CMPPS
- **Поразрядные логические операции**
 - Packed: ANDPS, ORPS, XORPS, ANDNPS
- ...

SSE-инструкции копирования данных

- **MOVSS:** Copy a single floating-point data
- **MOVLPS:** Copy 2 floating-point data (low packed)
- **MOVHPS:** Copy 2 floating-point data (high packed)
- **MOVAPS:** Copy aligned 4 floating-point data (fast)
- **MOVUPS:** Copy unaligned 4 floating-point data (slow)
- **MOVHLPS:** Copy 2 high elements to low position
- **MOVLHPS:** Copy 2 low elements to high position

При копировании данных из памяти в векторный регистр (и наоборот) рекомендуется чтобы адрес был выровнен на границу в 16 байт

SSE-инструкции копирования данных



Арифметические SSE-инструкции

Arithmetic	Scalar Operator	Packed Operator
$y = y + x$	addss	addps
$y = y - x$	subss	subps
$y = y \times x$	mulss	mulps
$y = y \div x$	divss	divps
$y = \frac{1}{x}$	rcpss	rcpps
$y = \sqrt{x}$	sqrts	sqrtps
$y = \frac{1}{\sqrt{x}}$	rsqrts	rsqrtps
$y = \max(y, x)$	maxss	maxps
$y = \min(y, x)$	minss	minps

Использование инструкций SSE

Ассемблерные
вставки

Встроенные
функции
компилятора
(Intrinsic)

C++ классы

Автоматическая
векторизация
компилятора



*Лучшая управляемость
(полный контроль)*

*Простота
использования*

Автовекторизация компилятором

- **Clang (LLVM)** – векторизация включена по умолчанию

```
$ clang -mllvm -force-vector-width=8 ...  
$ opt -loop-vectorize -force-vector-width=8 ...  
$ clang -fslp-vectorize-aggressive ./prog.c
```

- **Visual C++ 2012** – векторизация включена по умолчанию (при использовании опции /O2, подробный отчет формируется опцией /Qvec-report)
- **Intel C++ Compiler** – векторизация включена по умолчанию (при использовании опции /O2, -O2, подробный отчет формируется опцией /Qvec-report, -vec-report)
- **Oracle Solaris Studio** – векторизация включается совместным использованием опций -xvector=simd и -xO3

Автовекторизация компилятором

- **GNU GCC** – векторизация включается при использовании опции `-O3` или `-ftree-vectorize`
- Векторизация для PowerPC (набор инструкций AltiVec) включается опцией `-maltivec`
- Векторизация для ARM NEON: `-mfpu=neon -mfloat-abi=softfp` или `-mfloat-abi=hard`

<http://gcc.gnu.org/projects/tree-ssa/vectorization.html>

Автовекторизация компилятором

```
#define N 100

int main()
{
    int i, a[N];

    for (i = 0; i < N; i++) {
        a[i] = a[i] >> 2;
    }
    return 0;
}
```


Автовекторизация компилятором GCC 5.1.1

```
$ gcc -O2 -ftree-vectorize -msse4  
    --save-temps /prog.c
```

```
...  
.L2:  
    movdqa    (%rax), %xmm0  
    addq      $16, %rax  
    psrad     $2, %xmm0  
    movaps    %xmm0, -16(%rax)  
    cmpq      %rbp, %rax  
    jne       .L2  
...
```

Автовекторизация компилятором Intel

```
$ gcc -xP -o prog ./prog.c
```

```
#if defined (__INTEL_COMPILER)
#pragma vector always
#endif
for (i = 0; i < 100; i++) {
    k = k + 10;
    a[i] = k;
}
```

C++ классы SSE (Intel Compiler only)

```
void add(float *a, float *b, float *c)
{
    int i;

    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

C++ классы SSE (Intel Compiler only)

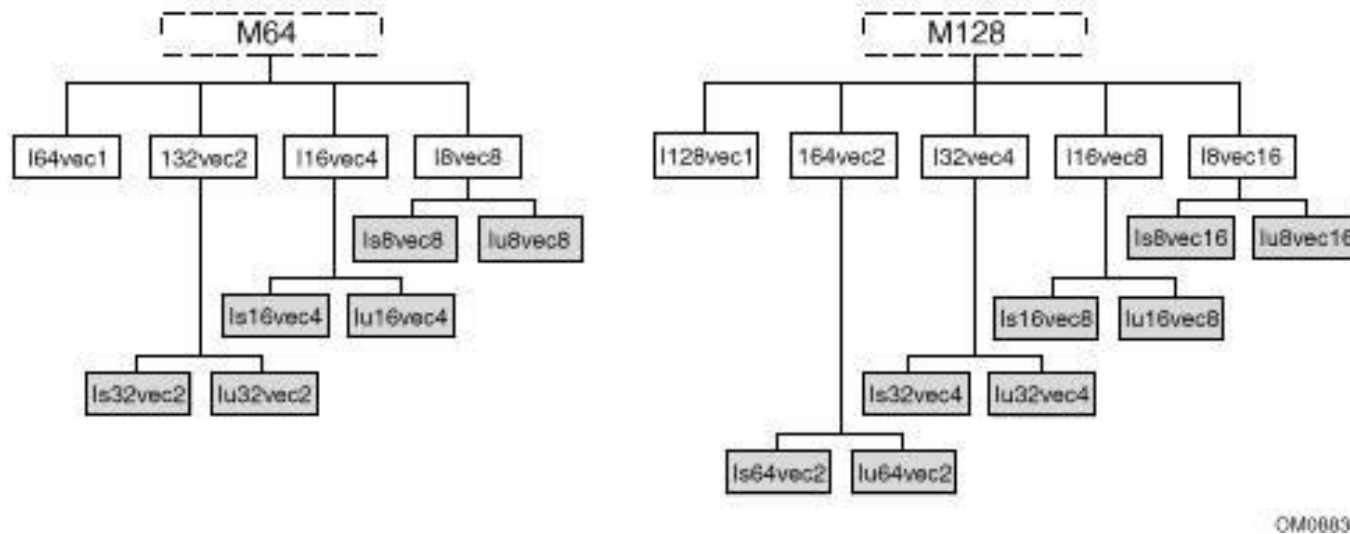
```
#include <fvec.h>    /* SSE classes */

void add(float *a, float *b, float *c)
{
    F32vec4 *av = (F32vec4 *)a;
    F32vec4 *bv = (F32vec4 *)b;
    F32vec4 *cv = (F32vec4 *)c;

    *cv = *av + *bv;
}
```

- **F32vec4** – класс, представляющий массив из 4 элементов типа float

C++ классы SSE (Intel Compiler only)



Classes Quick Reference

<https://software.intel.com/en-us/node/524434>

Вставки на ассемблере

```
void add_sse_asm(float *a, float *b, float *c)
{
    __asm__ __volatile__
    (
        "movaps (%[a]), %%xmm0 \n\t"
        "movaps (%[b]), %%xmm1 \n\t"
        "addps %%xmm1, %%xmm0 \n\t"
        "movaps %%xmm0, %[c] \n\t"
        : [c] "=m" (*c)           /* output */
        : [a] "r" (a), [b] "r" (b) /* input */
        : "%xmm0", "%xmm1"        /* modified regs */
    );
}
```

SSE Intrinsics (builtin functions)

- **Intrinsics** – набор встроенных функций и типов данных, поддерживаемых компилятором, для предоставления высокоуровневого доступа к SSE-инструкциям
- Компилятор самостоятельно распределяет XMM/YMM регистры, принимает решение о способе загрузки данных из памяти (проверяет выравнен адрес или нет) и т.п.
- **Заголовочные файлы:**

```
#include <mmintrin.h>    /* MMX */
#include <xmmintrin.h>    /* SSE, нужен также mmintrin.h */
#include <emmintrin.h>    /* SSE2, нужен также xmmintrin.h */
#include <pmmintrin.h>    /* SSE3, нужен также emmintrin.h */
#include <smmintrin.h>    /* SSE4.1 */
#include <nmmintrin.h>    /* SSE4.2 */
#include <immintrin.h>    /* AVX */
```

SSE Intrinsics: типы данных

```
void main()
{
    __m128  f;    /* float[4] */
    __m128d d;    /* double[2] */
    __m128i i;    /* char[16], short int[8], int[4],
                  uint64_t [2] */
}
```


SSE Intrinsics

- Названия Intrinsic-функций

`__mm_<intrinsic_name>_<suffix>`

```
void main()
{
    float v[4] = {1.0, 2.0, 3.0, 4.0};
    __m128 t1 = __mm_load_ps(v); // v must be 16-byte aligned

    __m128 t2 = __mm_set_ps(4.0, 3.0, 2.0, 1.0);
}
```

SSE Intrinsics

```
void add(float *a, float *b, float *c)
{
    int i;

    for (i = 0; i < 4; i++) {
        c[i] = a[i] + b[i];
    }
}
```

SSE Intrinsics

```
#include <xmmintrin.h>    /* SSE */

void add(float *a, float *b, float *c)
{
    __m128 t0, t1;

    t0 = _mm_load_ps(a);
    t1 = _mm_load_ps(b);
    t0 = _mm_add_ps(t0, t1);
    _mm_store_ps(c, t0);
}
```

Выравнивание памяти: Microsoft Windows

■ Выравнивание памяти

Хранимые в памяти операнды SSE-инструкций должны быть размещены по адресу выровненному на границу в 16 байт

```
/* Определение статического массива */
__declspec(align(16)) float A[N];

/*
 * Динамическое выделение памяти
 * с заданным выравниванием адреса
 */
#include <malloc.h>
void *_aligned_malloc(size_t size, size_t alignment);
void _aligned_free(void *memblock);
```

Выравнивание памяти: GNU/Linux

■ Выравнивание памяти

Хранимые в памяти операнды SSE-инструкций должны быть размещены по адресу выровненному на границу в 16 байт

```
/* Определение статического массива */
float A[N] __attribute__((aligned(16)));

/*
 * Динамическое выделение памяти
 * с заданным выравниванием адреса
 */
#include <malloc.h>
void *_mm_malloc(size_t size, size_t align)
void _mm_free(void *p)

#include <stdlib.h>
int posix_memalign(void **memptr, size_t alignment,
                  size_t size);
```

Функции копирования данных

```
#include <xmmintrin.h>    /* SSE */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
<code>__m128 _mm_load_ss(float * p)</code>	Load the low value and clear the three high values	MOVSS
<code>__m128 _mm_load1_ps(float * p)</code>	Load one value into all four words	MOVSS + Shuffling
<code>__m128 _mm_load_ps(float * p)</code>	Load four values, address aligned	MOVAPS
<code>__m128 _mm_loadu_ps(float * p)</code>	Load four values, address unaligned	MOVUPS
<code>__m128 _mm_loadr_ps(float * p)</code>	Load four values in reverse	MOVAPS + Shuffling

Функции копирования данных

```
#include <emmintrin.h>    /* SSE2 */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
<code>__m128d</code> <code>_mm_load_pd(double const *dp)</code>	Loads two DP FP values	MOVAPD
<code>__m128d</code> <code>_mm_load1_pd(double const *dp)</code>	Loads a single DP FP value, copying to both elements	MOVSD + shuffling
<code>__m128d</code> <code>_mm_loadr_pd(double const *dp)</code>	Loads two DP FP values in reverse order	MOVAPD + shuffling
<code>__m128d</code> <code>_mm_loadu_pd(double const *dp)</code>	Loads two DP FP values	MOVUPD
<code>__m128d</code> <code>_mm_load_sd(double const *dp)</code>	Loads a DP FP value, sets upper DP FP to zero	MOVSD

Функции копирования данных

t

4.0	3.0	2.0	1.0
-----	-----	-----	-----

```
t = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
```

t

1.0	1.0	1.0	1.0
-----	-----	-----	-----

```
t = _mm_set1_ps(1.0);
```

t

0.0	0.0	0.0	1.0
-----	-----	-----	-----

```
t = _mm_set_ss(1.0);
```

t

0.0	0.0	0.0	0.0
-----	-----	-----	-----

```
t = _mm_setzero_ps();
```


Арифметические операции

```
#include <xmmintrin.h>    /* SSE */
```

Intrinsic Name	Operation	Corresponding SSE Instruction
__m128 _mm_add_ss(__m128 a, __m128 b)	Addition	ADDSS
_mm_add_ps	Addition	ADDPS
_mm_sub_ss	Subtraction	SUBSS
_mm_sub_ps	Subtraction	SUBPS
_mm_mul_ss	Multiplication	MULSS
_mm_mul_ps	Multiplication	MULPS
_mm_div_ss	Division	DIVSS
_mm_div_ps	Division	DIVPS
_mm_sqrt_ss	Squared Root	SQRTSS
_mm_sqrt_ps	Squared Root	SQRTPS
_mm_rcp_ss	Reciprocal	RCPSS
_mm_rcp_ps	Reciprocal	RCPPS
_mm_rsqrt_ss	Reciprocal Squared Root	RSQRTSS
_mm_rsqrt_ps	Reciprocal Squared Root	RSQRTPS
_mm_min_ss	Computes Minimum	MINSS
_mm_min_ps	Computes Minimum	MINPS
_mm_max_ss	Computes Maximum	MAXSS
_mm_max_ps	Computes Maximum	MAXPS

Арифметические операции

```
#include <emmintrin.h>    /* SSE2 */
```

Intrinsic Name	Operation	Corresponding Intel® SSE Instruction
__m128d _mm_add_sd(__m128d a, __m128d b)	Addition	ADDSD
_mm_add_pd	Addition	ADDPD
_mm_sub_sd	Subtraction	SUBSD
_mm_sub_pd	Subtraction	SUBPD
_mm_mul_sd	Multiplication	MULSD
_mm_mul_pd	Multiplication	MULPD
_mm_div_sd	Division	DIVSD
_mm_div_pd	Division	DIVPD
_mm_sqrt_sd	Computes Square Root	SQRTSD
_mm_sqrt_pd	Computes Square Root	SQRTPD
_mm_min_sd	Computes Minimum	MINSD
_mm_min_pd	Computes Minimum	MINPD
_mm_max_sd	Computes Maximum	MAXSD
_mm_max_pd	Computes Maximum	MAXPD

SSE Intrinsics

- **Intel® C++ Compiler XE 13.1 User and Reference Guide (Intrinsics for SSE{2, 3, 4}), AVX** // <http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/GUID-712779D8-D085-4464-9662-B630681F16F1.htm>
- **GCC 4.8.1 Manual (X86 Built-in Functions)** // http://gcc.gnu.org/onlinedocs/gcc-4.8.1/gcc/X86-Built_002din-Functions.html#X86-Built_002din-Functions
- **Clang API Documentation** // http://clang.llvm.org/doxygen/emmintrin_8h_source.html

Пример 1

```
void fun(float *a, float *b, float *c, int n)
{
    int i;

    for (i = 0; i < n; i++) {
        c[i] = sqrt(a[i] * a[i] +
                    b[i] * b[i]) + 0.5f;
    }
}
```

Пример 1 (SSE)

```
void fun_sse(float *a, float *b, float *c, int n)
{
    int i, k;
    __m128 x, y, z;
    __m128 *aa = (__m128 *)a;
    __m128 *bb = (__m128 *)b;
    __m128 *cc = (__m128 *)c;

    /* Предполагаем, что n кратно 4 */
    k = n / 4;
    z = _mm_set_ps1(0.5f);
    for (i = 0; i < k; i++) {
        x = _mm_mul_ps(*aa, *aa);
        y = _mm_mul_ps(*bb, *bb);
        x = _mm_add_ps(x, y);
        x = _mm_sqrt_ps(x);
        *cc = _mm_add_ps(x, z);
        aa++;
        bb++;
        cc++;
    }
}
```

Пример 1 (Benchmarking)

```
enum { N = 1024 * 1024, NREPS = 10 };

/* Implementation ... */

int main(int argc, char **argv)
{
    int i;
    float *a, *b, *c;
    double t;

    a = (float *)_mm_malloc(sizeof(float) * N, 16);
    b = (float *)_mm_malloc(sizeof(float) * N, 16);
    c = (float *)_mm_malloc(sizeof(float) * N, 16);
    for (i = 0; i < N; i++) {
        a[i] = 1.0; b[i] = 2.0;
    }

    t = hpctimer_getwtime();
    for (i = 0; i < NREPS; i++)
        fun_sse(a, b, c, N); /* fun(a, b, c, N); */
    t = (hpctimer_getwtime() - t) / NREPS;
    printf("Elapsed time: %.6f sec.\n", t);

    _mm_free(a); _mm_free(b); _mm_free(c);
    return 0;
}
```

Пример 1 (Benchmarking)

- Intel Core 2 i5 2520M (Sandy Bridge)
- GNU/Linux (Fedora 19) x86_64 kernel 3.10.10-200.fc19.x86_64
- GCC 4.8.1
- Флаги компиляции: `gcc -Wall -O2 -msse3 -o vec ./vec.c`

Функция	Время, сек.	Ускорение (Speedup)
fun (исходная версия)	0.007828	—
fun_sse (SSE2 Intrinsics)	0.001533	5.1

AVX Intrinsics

```
__m256  f;    /* float[8] */
__m256d d;    /* double[4] */
__m256i i;    /* char[32], short int[16], int[8],
               uint64_t [4] */
```


Пример 1 (AVX)

```
void fun_avx(float *a, float *b, float *c, int n)
{
    int i, k;
    __m256 x, y;
    __m256 *aa = (__m256 *)a;
    __m256 *bb = (__m256 *)b;
    __m256 *cc = (__m256 *)c;

    k = n / 8;
    for (i = 0; i < k; i++) {
        x = _mm256_mul_ps(*aa, *aa);
        y = _mm256_mul_ps(*bb, *bb);
        x = _mm256_add_ps(x, y);
        *cc = _mm256_sqrt_ps(x);
        aa++;
        bb++;
        cc++;
    }
}
```

Пример 2

```
void shift(int *v, int n)
{
    int i;

    for (i = 0; i < n; i++) {
        v[i] = v[i] >> 2;
    }
}
```

Пример 2 (SSE)

```
void shift_sse(int *v, int n)
{
    int i;
    __m128i *i4 = (__m128i *)v;

    /* Полагаем, n – кратно 4 */
    for (i = 0; i < n / 4; i++) {
        i4[i] = _mm_srai_epi32(i4[i], 2);
    }
}
```

- `__m128i _mm_srai_epi32(__m128i a, int count);`
- Shifts the 4 signed 32-bit integers in a right by count bits

```
r0 := a0 >> count;    r1 := a1 >> count
r2 := a2 >> count;    r3 := a3 >> count
```

Пример 2 (SSE)

```
void shift_sse(int *v, int n)
{
    int i;
    __m128i *i4 = (__m128i *)v;

    /* Полагаем, n – кратно 4 */
    for (i = 0; i < n / 4; i++) {
        i4[i] = _mm_srai_epi32(i4[i], 2);
    }
}
```

Speedup 1.65 (65%)

- Intel Core i5 2520M (Sandy Bridge)
- Linux x86_64 (Fedora 19)
- GCC 4.8.1, opt. flags: -O2 -msse3
- $n = 16 * 1024 * 1024$

Пример 3: Reduction/Summation

```
float reduction(float *v, int n)
{
    int i;
    float sum = 0.0;

    for (i = 0; i < n; i++) {
        sum += v[i];
    }
    return sum;
}
```

Пример 3: SSE3 Reduction/Summation

```
#include <pmmmintrin.h>    /* SSE3 */

float reduction_sse(float *v, int n)
{
    int i;
    float sum;
    __m128 *v4 = (__m128 *)v;
    __m128 vsum = _mm_set1_ps(0.0f);

    for (i = 0; i < n / 4; i++)
        vsum = _mm_add_ps(vsum, v4[i]);

    /* Horizontal sum: | a3+a2 | a1+a0 | a3+a2 | a1+a0 | */
    vsum = _mm_hadd_ps(vsum, vsum);
    /* Horizontal sum: | a3+a2+a1+a0 | -- | -- | -- | */
    vsum = _mm_hadd_ps(vsum, vsum);
    _mm_store_ss(&sum, vsum);
    return sum;
}
```

Пример 3: SSE3 Reduction/Summation

```
#include <pmmmintrin.h>    /* SSE3 */

float reduction_sse(float *v, int n)
{
    int i;
    float sum;
    __m128 *v4 = (__m128 *)v;
    __m128 vsum = _mm_set1_ps(0.0f);

    for (i = 0; i < n / 4; i++)
        vsum = _mm_add_ps(vsum, v4[i]);

    /* Horizontal sum: | a3+a2 | a1+a0 | a3+a2 | a1+a0 | */
    vsum = _mm_hadd_ps(vsum, vsum);
    sum = vsum[0];
}
```

Speedup 3.2 (220%)

- Intel Core i5 2520M (Sandy Bridge)
- Linux x86_64 (Fedora 19)
- GCC 4.8.1, opt. flags: -O2 -msse3
- $n = 16 * 1024 * 1024$

Пример 3: AVX Reduction/Summation

```
#include <immintrin.h>    /* AVX */

float reduction_avx(float *v, int n)
{
    int i;
    float vres[8] __attribute__((aligned(32)));
    __m256 *v8 = (__m256 *)v;
    __m256 vsum = _mm256_setzero_ps();

    for (i = 0; i < n / 8; i++)
        vsum = _mm256_add_ps(vsum, v8[i]);

    /* Horizontal summation */
    vsum = _mm256_hadd_ps(vsum, vsum);
    vsum = _mm256_hadd_ps(vsum, vsum);
    vsum = _mm256_hadd_ps(vsum, vsum);

    _mm256_store_ps(vres, vsum);
    return vres[0];
}
```


Пример 3: AVX Reduction/Summation

```
#include <immintrin.h>    /* AVX */

float reduction_avx(float *v, int n)
{
    int i;
    float vres[8] __attribute__((aligned(32)));
    __m256 *v8 = (__m256 *)v;
    __m256 vsum = _mm256_setzero_ps();

    for (i = 0; i < n / 8; i++)
        vsum = _mm256_add_ps(vsum, v8[i]);

    /* Horizontal summation */
    vsum = _mm256_hadd_ps(vsum, vsum);
}
```

Speedup 3.67 (267%)

- Intel Core i5 2520M (Sandy Bridge)
- Linux x86_64 (Fedora 19)
- GCC 4.8.1, opt. flags: -O2 -mavx
- $n = 16 * 1024 * 1024$

Пример 4: Max

```
float vmax(float *v, int n)
{
    int i;
    float maxval = 0.0;

    for (i = 0; i < n; i++) {
        if (v[i] > maxval)
            maxval = v[i];
    }
    return maxval;
}
```

Пример 4: Max (SSE)

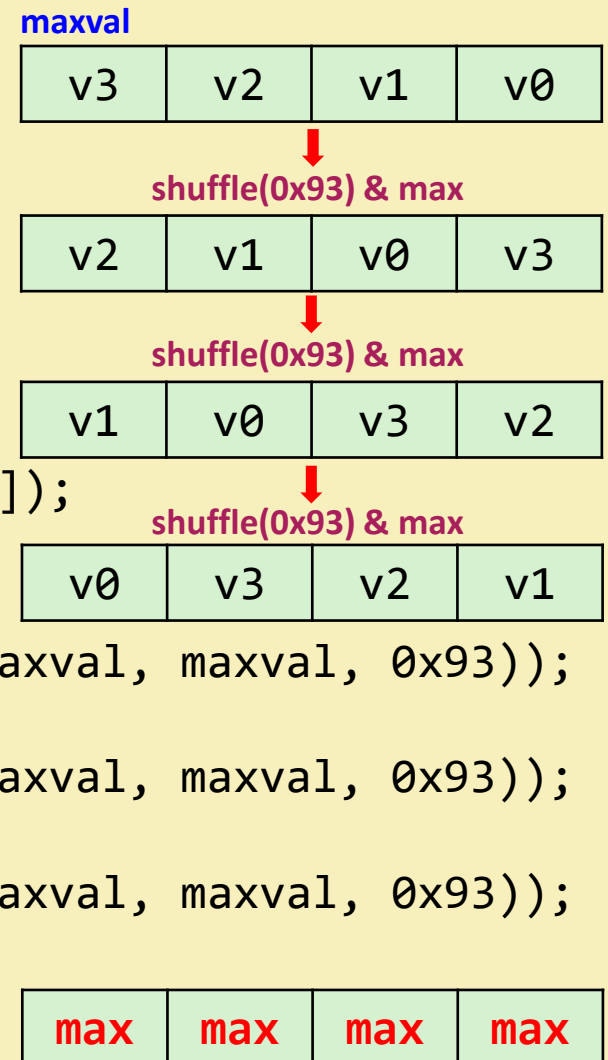
```
float vmax_sse(float *v, int n)
{
    int i;
    float res;
    __m128 *f4 = (__m128 *)v;
    __m128 maxval = _mm_setzero_ps();

    for (i = 0; i < n / 4; i++)
        maxval = _mm_max_ps(maxval, f4[i]);
    /* Horizontal max */
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    _mm_store_ss(&res, maxval);
    return res;
}
```

Пример 4: Max (SSE)

```
float vmax_sse(float *v, int n)
{
    int i;
    float res;
    __m128 *f4 = (__m128 *)v;
    __m128 maxval = _mm_setzero_ps();

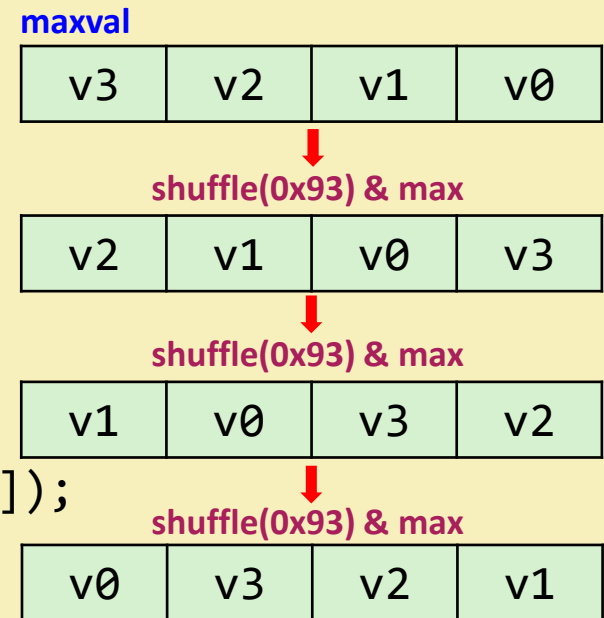
    for (i = 0; i < n / 4; i++)
        maxval = _mm_max_ps(maxval, f4[i]);
    /* Horizontal max */
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    _mm_store_ss(&res, maxval);
    return res;
}
```



Пример 4: Max (SSE)

```
float vmax_sse(float *v, int n)
{
    int i;
    float res;
    __m128 *f4 = (__m128 *)v;
    __m128 maxval = _mm_setzero_ps();
```

```
    for (i = 0; i < n / 4; i++)
        maxval = _mm_max_ps(maxval, f4[i]);
    /* Horizontal max */
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
    maxval = _mm_max_ps(maxval,
                        _mm_shuffle_ps(maxval, maxval, 0x93));
```



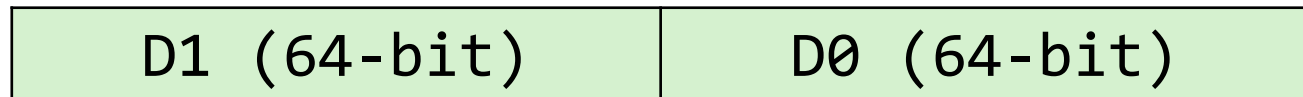
Speedup 4.28 (328%)

- Intel Core i5 2520M (Sandy Bridge)
- Linux x86_64 (Fedora 19)
- GCC 4.8.1, opt. flags: -O2 -msse3
- $n = 16 * 1024 * 1024$

ARM NEON SIMD Engine

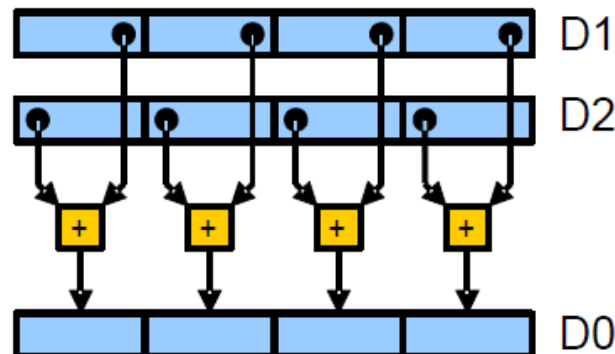
- **Векторные регистры:** 64 или 128 бит
- **Типы данных:** signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single precision floating point

Register Q0 (128-bit)



- Сложение двух 16-битных целых: $D0 = D1 + D2$

VADD.I16 D0, D1, D2



ARM NEON SIMD engine

```
#include <arm_neon.h>

int sum_array(int16_t *array, int size)
{
    /* Init the accumulator vector to zero */
    int16x4_t vec, acc = vdup_n_s16(0);
    int32x2_t acc1;
    int64x1_t acc2;

    for (; size != 0; size -= 4) {
        /* Load 4 values in parallel */
        vec = vld1_s16(array);
        array += 4;
        /* Add the vec to the accum vector */
        acc = vadd_s16(acc, vec);
    }
    acc1 = vpaddl_s16(acc);
    acc2 = vpaddl_s32(acc1);
    return (int)vget_lane_s64(acc2, 0);
}
```

Анализ SSE/AVX-программ

- **Intel Software Development Emulator** – эмулятор будущих микроархитектур и наборов команд Intel (Intel AVX-512, Intel SHA, Intel MPX, ...)
- **Intel Architecture Code Analyzer** – позволяет анализировать распределение инструкций по портам исполнительных устройств ядра процессора

Анализ SSE/AVX-программ

```
void fun_avx(float *a, float *b, float *c, int n)
{
    int i, k;
    __m256 x, y;
    __m256 *aa = (__m256 *)a;
    __m256 *bb = (__m256 *)b;
    __m256 *cc = (__m256 *)c;

    k = n / 8;
    for (i = 0; i < k; i++) {
        IACA_START
        x = _mm256_mul_ps(aa[i], aa[i]);
        y = _mm256_mul_ps(bb[i], bb[i]);
        x = _mm256_add_ps(x, y);
        cc[i] = _mm256_sqrt_ps(x);
        IACA_END
    }
}
```

Анализ SSE/AVX-программ

```
void fun_avx(float *a, float *b, float *c, int n)
{
    int i, k;
    __m256 x, y;
    __m256 *aa = (__m256 *)a;
    __m256 *bb = (__m256 *)b;
    __m256 *cc = (__m256 *)c;

    k = n / 8;
    for (i = 0; i < k; i++) {
        IACA_START
        x = _mm256_mul_ps(aa[i], aa[i]);
```

```
$ gcc -O2 -mavx -I ~/opt/iaca-lin32/include ./vec.c
...
```

```
$ iaca.sh -64 -arch SNB -analysis LATENCY \
    -graph ./mygraph ./vec
```

Intel IACA report

Latency Analysis Report

Latency: 59 Cycles

...

Inst	Resource Delay In Cycles									
Num	0 - DV	1	2 - D	3 - D	4	5	FE			
0	1								vmovaps ymm0, y	
1									vmulps ymm2, ym	
2								CP	vmovaps ymm0, y	
3								CP	vmulps ymm1, ym	
4								CP	vaddps ymm0, ym	
5								CP	vsqrtps ymm0, y	
6							2	CP	vmovaps ymmword	

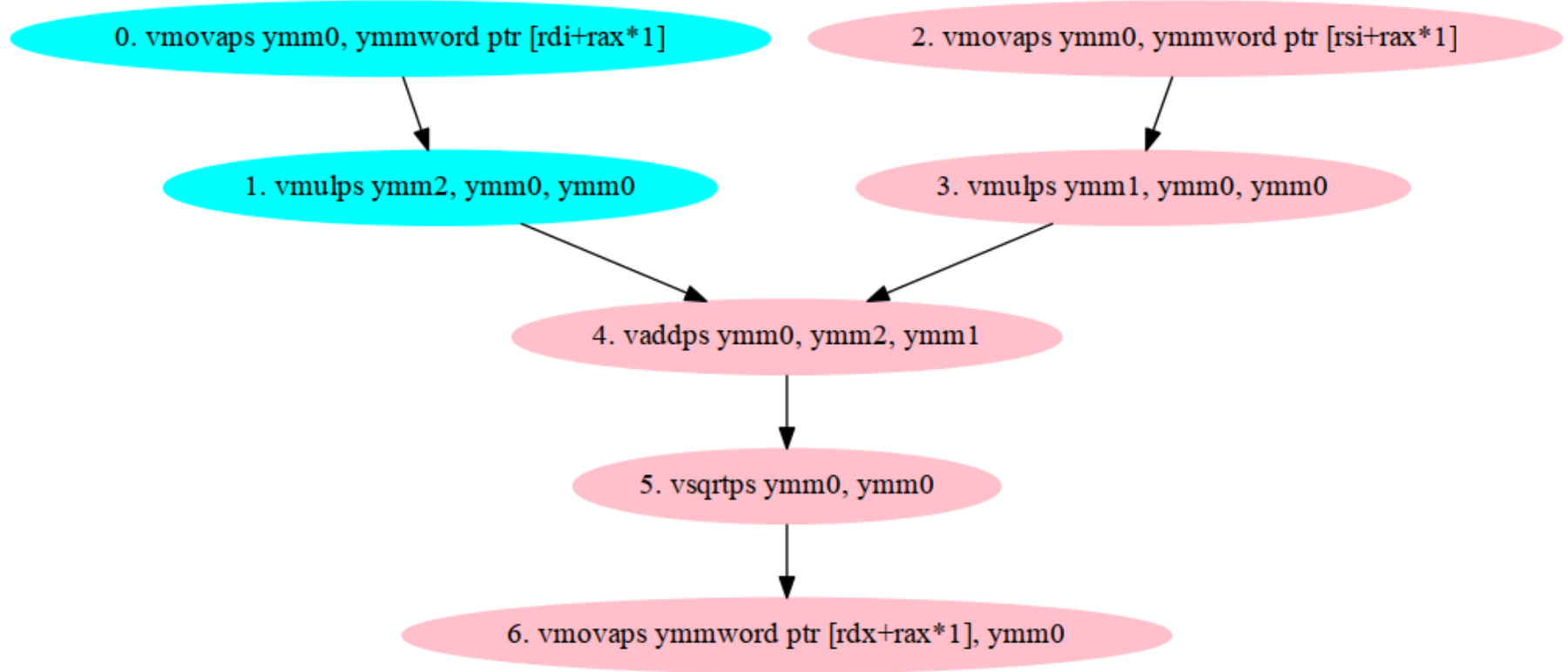
Resource Conflict on Critical Paths:

Port	0 - DV	1	2 - D	3 - D	4	5
Cycles	1 0	0	0 0	0 0	0	0

List Of Delays On Critical Paths

1 --> 3 1 Cycles Delay On Port

Intel IACA graph (data dependency)



Литература

- **Intel SSE4 Programming Reference //**
http://software.intel.com/sites/default/files/m/9/4/2/d/5/17971-intel_20sse4_20programming_20reference.pdf
- **Intel 64 and IA-32 Architectures Software Developer's Manual**
(Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B and 3C) //
<http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>
- **Intel Architecture Instruction Set Extensions Programming Reference //**
<http://download-software.intel.com/sites/default/files/319433-015.pdf>
- **A Guide to Vectorization with Intel C++ Compilers //** <http://download-software.intel.com/sites/default/files/m/d/4/1/d/8/CompilerAutovectorizationGuide.pdf>
- **AMD 128-Bit SSE5 Instruction Set //**
http://developer.amd.com/wordpress/media/2012/10/AMD64_128_Bit_SSE5_Instrs.pdf