Сравнительный анализ автоматического использования различных наборов инструкций x86 SIMD в задачах моделирования и обработки компьютерной графики

Копылов М.С., **Бирюков Е.Д.**, ИПМ им. М.В. Келдыша РАН
kopylov@gin.keldysh.ru, birukov@gin.keldysh.ru
Копылова Л.В., ФГБУ "46 ЦНИИ" Минобороны России
lvkopylova@hotmail.com

Аннотация

В работе рассматриваются аспекты использования расширенных наборов SIMD инструкций на современных 64-битных процессорах архитектуры Intel x86. Кратко перечисляются различные способы, позволяющие задействовать эти наборы инструкций при разработке и оптимизации приложений моделирования и обработки компьютерной графики. Производится анализ достигнутого ускорения при автоматическом использовании тех или иных наборов инструкций, а также эффективность автовекторизации на примере компилятора Microsoft C++ версии 14.1 (Visual Studio 2017).

1 Введение

Single Instruction Multiple Data (SIMD) – это модель параллельных вычислений, позволяющая обрабатывать большее количество данных за то же число инструкций процессора. Одной из наиболее важных причин, приведших к внедрению данной модели в современные процессорные архитектуры, являются резко возросшие требования графических приложений. В основном, это связано с тем, что использование SIMD наиболее актуально в задачах оперирующих большими объёмами однотипных данных, таких как различные массивы - пикселей, вершин, треугольников и т.д.

В настоящее время в архитектуре Intel x86 существует около десятка различных SIMD расширений. Они отличаются как типами поддерживаемых данных, способами работы с этими данными, так и наборами команд. При этом каждое последующее SIMD расширение является инкрементальным, т.е. включает в себя все предыдущие, благодаря чему достигается программная совместимость с ранее разработанными приложениями.

Самым первым набором SIMD расширений, появившимся в архитектуре Intel x86, стал набор команд MMX [Intel, 2013]. Данный набор из 57 новых команд позволяет обрабатывать целочисленные векторы с размером элемента 8, 16 и 32 бит. Для хранения этих векторов были добавлены восемь специальных 64-битных регистра `mm0, mm1, …, mm7`, которые, в свою очередь, являются ссылками на физические регистры стека арифметического сопроцессора x87 FPU. Подобная архитектурная реализация приводит к одному существенному недостатку, возникающему при использовании данного набора команд – невозможности совмещать целочисленные вычисления и вычисления с плавающей запятой, что требует от разработчиков программного обеспечения отдавать предпочтение командам конкретного типа.

Последующие наборы SIMD расширений – SSE, SSE2, SSE3, SSSE3, SSE 4.1, SSE 4.2, SSE4a – лишены этого недостатка, так как используют независимый регистровый файл, состоящий из шестнадцати (восьми в 32-битном режиме) 128-битных векторных регистров `xmm0, xmm1, …, xmm15`. Появились и новые типы поддерживаемых данных, такие как векторы типа `float`, состоящие из четырёх вещественных чисел одинарной точности, векторы типа `double`, состоящие из двух вещественных чисел двойной точности, а также целочисленные типы с размером элемента от 8 до 128 бит. Существенно расширился и набор команд – он стал включать в себя более двухсот новых инструкций, таких как команды пересылки, арифметические и логические операции, команды сравнения и преобразования типов, побитовые операции, горизонтальные операции над данными, хранящимися в одном регистре и т.д.

Наиболее современными SIMD расширениями, появившимися сравнительно недавно, стали наборы команд AVX, AVX2, FMA3, FMA4, XOP. Их главной особенностью стало увеличение размера векторных регистров до 256 бит, одновременно с этим они получили и новые имена `ymm0, ymm1, …, ymm15`. Уже

существующие 128-битные регистры `xmmn`, при этом стали являться ссылками на младшие половины регистров `ymmn`. Другим важным новшеством этих наборов команд стал трёхоперандный синтаксис кодирования инструкций, что привело к сильному сокращению размера машинного кода, а, следовательно, и увеличению быстродействия за счёт устранения использования лишних команд пересылки [Jain, T. and Agrawal, T., 2013].

В настоящее время вышеперечисленные наборы SIMD расширений поддерживаются всеми современными процессорами архитектуры Intel x86. Учитывая этот факт, в данной работе будет предпринята попытка оценить эффективность использования данных расширений в уже существующих программах.

2 Способы использования расширенных наборов инструкций

Существует несколько основных способов, позволяющих задействовать вышеперечисленные SIMD расширения в разрабатываемых или оптимизируемых программах [Pohl, A., Cosenza, B., Mesa, M.A., Chi, C.C. and Juurlink, B., 2016]. К числу этих способов можно отнести (в порядке усложнения использования):

1. Автоматическое использование расширенных наборов команд компилятором;
2. Использование различных библиотек классов, например Intel C++ SIMD Classes;
3. Использование встроенных функций компилятора (Intrinsic);
4. Использование ассемблерных вставок.

Выбор конкретного способа зависит от нескольких факторов, таких, как типы обрабатываемых программой данных, наличие определённых требований к скорости вычислений, наличие времени на проведение углублённых оптимизаций и т.д.

Важно отметить одну существенную особенность первого способа, отличающую его от трёх последующих – он не требует вносить какие-либо изменения в исходный код программы, необходимо лишь только перекомпилировать её. По этой причине в данной статье будет рассмотрен именно этот способ. Для проверки его эффективности будут проведены замеры прироста производительности, как в тестовых, так и в реальных задачах на примере комплекса оптического моделирования разрабатываемого в ИПМ им. М. В. Келдыша РАН.

3 Автоматическое использование SIMD расширений компилятором

Современные компиляторы используют алгоритмы анализа кода, позволяющие проводить оптимизацию на этапе трансляции в машинные команды. Существует два основных вида подобных оптимизаций:

- принудительное использование расширенных наборов команд (SIMD расширений);
- автовекторизация.

Прежде чем переходить к детальному рассмотрению вышеперечисленных способов, необходимо рассмотреть одну важную особенность SIMD расширений, такую, как возможность производить вычисления не только над **векторными (packed)**, но и над **скалярными (scalar)** данными целых и вещественных типов. Так, большинство векторных SSE, SSE2, AVX и AVX2 команд имеют специальные скалярные аналоги, в которых одним из операндов является не весь векторный регистр, а лишь его младший элемент. Благодаря поддержке скалярных вычислений, SIMD расширения могут использоваться в качестве альтернативных наборов команд, заменяющих собой большинство команд сопроцессора x87 FPU при работе с вещественными числами, а также определённую часть целочисленных команд общего назначения. Также важно отметить тот факт, что арифметические и логические блоки современных процессоров, выполняющие SIMD инструкции, имеют существенно более высокое быстродействие, по сравнению с блоками общего назначения ALU, и в особой степени x87 FPU, который из-за своей стековой структуры осложняет эффективную работу с несколькими операндами.

Первый вид оптимизаций основан на использовании только **скалярных** команд SIMD. Он предписывает компилятору задействовать скалярные инструкции, входящие в предписанный набор SIMD расширений, везде, где это представляется возможным.

Для включения данного способа в компиляторах Microsoft C++ используется специальный ключ `/arch`, который может принимать одно из следующих значений - `IA32`, `SSE`, `SSE2`, `AVX`, `AVX2`, что подразумевает активацию выбранного набора команд (значение `IA32` запрещает использования каких либо команд кроме команд общего назначения и инструкций x87 FPU).

Автовекторизация является продолжением первого вида оптимизаций и основана на использовании как **скалярных,** так и **векторных** команд SIMD. В этом случае компилятор будет дополнительно пытаться преобразовать скалярный код в векторный, например, раскручивая циклы с целью выполнить сразу несколько итераций с помощью одной векторной команды. Для активации данного способа в компиляторах Microsoft C++ должен быть задействован ключ `/O2` в паре с ключом `/arch`.

Важно отметить, что компилятор не добавляет в генерируемый машинный код явной проверки на наличие поддержки выбранного набора команд процессором, поэтому оба способа не могут считаться достаточно гибкими. Попытка выполнить скомпилированное приложение на неподдерживаемом процессоре приведёт к генерированию исключения `EXCEPTION_ILLEGAL_INSTRUCTION`.

В Табл. 1. приводятся краткие сравнительные характеристики расширенных наборов команд доступные при работе с векторными и скалярными операндами. Учитывая эти характеристики, а также описанные выше виды оптимизаций возникает предположение, что перекомпиляция существующих и разрабатываемых программ с их учётом может существенно повысить их быстродействие. При этом наибольший прирост будет наблюдаться в тех программах, которые работают с большими массивами данных. Например, в задачах оптического моделирования наиболее часто используемым типом данных является вещественное число двойной точности `double`, поэтому перекомпиляция таких приложений с ключами `/arch=SSE2` (или `/arch=AVX`) и `/O2`, видится авторам наиболее логичным шагом при оптимизации подобных программ.

Табл. 1. Размеры поддерживаемых операндов (бит)

	IA32	SSE	SSE2	AVX	AVX2
Integer scalar	до 64	н/п	до 64	до 64	до 64
Integer vector	н/п	н/п	16*8, 8*16, 4*32, 2*64	16*8, 8*16, 4*32, 2*64	32*8, 16*16, 8*32, 4*64
Float scalar	32, 64, 80	32	32, 64	32, 64	32, 64
Float vector	н/п	4*32	4*32 2*64	8*32 4*64	8*32 4*64

Для проверки данной гипотезы и вышеперечисленных автоматических видов оптимизаций был проведён эксперимент, включающий в себя использование специального бенчмарка Flops [Flops benchmark]. Этот бенчмарк представляет собой простую программу с открытым исходным кодом, написанную на языке C++, которая выполняет операции над вещественными числами двойной точности и позволяет измерить количество таких операций, которое способна выполнить та или иная система за определённый период времени. Основные вычисления выполняются в теле цикла, как показано ниже:

```
register double op1;
register double op2;
register double op3;
double res;
...
for (; cnt < n_it; cnt++)
  {
  double t;
  res = t + (op1 * op2) + op3;
  }
...
```

В первую очередь производился замер прироста скалярной производительности в зависимости от выбранных наборов инструкций. Бенчмарк был скомпилирован:

- без использования расширенных наборов команд (`/arch=IA32`);
- с использованием инструкций SSE2 (`/arch=SSE2`);
- с использованием инструкций AVX (`/arch=AVX`);

Для всех вышеперечисленных случаев выполнялось дисассемблирование исполняемого файла. Ниже приводятся листинги машинного кода, соответствующие упрощённому телу цикла бенмарка.

- для `/arch=IA32`, используются команды сопроцессора x87 FPU

```
...
fld [ebp-40h]
fstp [ebp-58h]
fld [op1]
fmul [ebp-48h]
fadd [ebp-50h]
fadd [ebp-58h]
fstp [res]
...
```

- для `/arch=SSE2`, используются скалярные SSE команды

```
...
```

```
mulsd xmm0, xmm1
addsd xmm0, xmm2
addsd xmm0, [rbp-8h]
movsd [res], xmm0
…
```

- для /arch=AVX, используются скалярные AVX команды

```
…
vmulsd xmm0, xmm0, xmm1
vaddsd xmm0, xmm0, xmm2
vaddsd [res], xmm0, [rbp-8h]
…
```

Данные листинги подтверждают использование дополнительных SIMD расширений при компиляции приложений с ключом /arch, отличным от IA32, для скалярного кода.

На Рис. 1. приведены результаты прироста быстродействия, полученные для современных 64-разрядных x86 процессоров от фирм Intel и AMD. Процессоры с архитектурами Core 2 от Intel и K10 от AMD не поддерживают расширения AVX, поэтому их результаты доступны только для SSE2.

Средний прирост быстродействия при выполнении бенчмарка составил от 15% до 51%, при этом наибольшее ускорение достигается на самых современных архитектурах процессоров – Kaby Lake от Intel и K17 Zen от AMD.

Так же заметен чуть меньший, но весьма существенный прирост производительности при переходе от SSE2 к AVX. Это связано с двумя факторами. В первую очередь, начиная с набора команд AVX, требование, по которому операнды SIMD инструкций всегда должны размещаться по адресу, выровненному на границу в 16 байт, стало необязательным [Jain, T. and Agrawal, T., 2013]. Во вторых, инструкции AVX и AVX2 используют трёхоперандный (non-destructive) синтаксис команд. Он позволяет сохранять значения операндов-источников, а результаты записывать в новые операнды-приёмники. Благодаря этому, машинный код получается гораздо меньшего размера за счёт устранения лишних команд пересылки данных, что подтверждается приведенным выше листингом для /arch=AVX.

На Рис. 2. приведены результаты прироста быстродействия после того, как бенчмарк был скомпилирован с включённым режимом автовекторизации /O2 для всех возможных значений ключа /arch.

Как видно из результатов, наибольшую эффективность демонстрирует код, скомпилированный с использованием расширений SSE2, AVX и AVX, прирост быстродействия при этом может быть четырёхкратным.

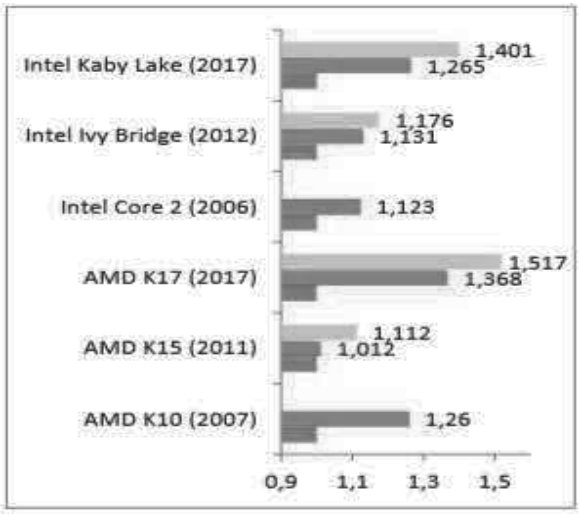

Рис. 1. Прирост производительности при выполнении операций с вещественных числами двойной точности FP64 при использовании скалярных вычислений на SSE2 (красный) и AVX (зеленый) относительно FPU (синий)

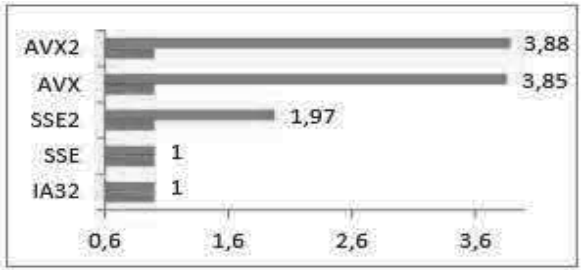

Рис. 2. Относительный прирост производительности при выполнении операций с вещественными числами двойной точности FP64 при включении автовекторизации (опция /O2)

4 Вычислительный эксперимент

Учитывая положительные результаты, полученные от задействования расширенных наборов инструкций и автовекторизации, было решено оценить эффективность подобных оптимизаций в реальных условиях в рамках работы с комплексом оптического моделирования разрабатываемого в ИПМ им. М. В. Келдыша РАН.

Для этого комплекс был последовательно перекомпилирован с использованием различных сочетаний ключей компилятора /arch и /O2. В качестве критерия оценки использовалось время, затраченное на выполнение опре-

делённых расчётов над сценами, загруженными в ядро комплекса. Данные расчёты включали в себя:

1) Рендеринг геометрической модели методом Backward ray-tracing (BRT);
2) Применение медианного фильтра [Бирюков, 2014] к графическому изображению;
3) Преобразование геометрической модели из одного формата в другой.

Выбор вышеперечисленных видов расчётов был обусловлен тем, что они в основном оперируют данными представляющими из себя наборы векторов, матриц и линейных массивов, при работе с которыми прирост быстродействия от автовекторизации должен проявляться наиболее сильно.

Вычислительный эксперимент был осуществлён на ПК со следующими характеристиками: Процессор: Intel Core i3-7300 4,0 ГГц; RAM: 32 Гб; ОС: Windows 10 x64. В качестве компилятора использовался Microsoft C++ Compiler версии 14.1 (Visual Studio 2017). Сравнение замеров производительности приведено в Табл. 2.

Как видно из результатов эксперимента, использование расширенных наборов инструкций положительным образом сказывается на быстродействии комплекса оптического моделирования; наибольший эффект достигается при задействовании наборов инструкций SSE2, AVX, AVX2, при этом значения относительно прироста оказываются весьма близкими к значениям прироста достигнутым ранее в бенчмерке Flops и составляют от 10% до 29%.

Относительный прирост от задействования автовекторизации оказался в разы ниже полученного в идеальных условиях и варьировался от 15% при задействовании инструкций SSE2 до 34% в AVX2 соответственно. Общий прирост составил в среднем около 25%, что в любом случае является очень хорошим результатом, учитывая ту легкость, с которой он был достигнут.

Табл. 2. Результаты замеров производительности при использовании автовекторизации с различными наборами SIMD расширений

Расчёт	Автовекторизация	IA32	SSE	SSE2	AVX	AVX2
1	Disabled /Od	151.7 с	150.9 с	137.4 с	133.1 с	129.3 с
	/O2	151.5 с	148.7 с	131.1 с	125.0 с	122.4 с
2	Disabled /Od	312.8 с	307.1 с	274.8 с	269.2 с	255.9 с
	/O2	310.2 с	306.9 с	267.5 с	262.7 с	246.1 с
3	Disabled /Od	542.0 с	541.1 с	472.7 с	423.4 с	417.6 с
	/O2	539.9 с	539.7 с	443.0 с	406.8 с	402.3 с

5 Заключение

При выполнении данной работы были сделаны следующие выводы:

- Автоматическое использование расширенных наборов инструкций и автовекторизации компилятором положительно сказывается на быстродействии программ. Наиболее ощутимый прирост скорости достигается при активации обоих этих способов.
- В реальных задачах, эффективность автовекторизации в разы меньше её теоретического максимума, который наблюдался выполнении специального бенчмарка.
- Использование современных наборов команд AVX и AVX2 является более предпочтительным, чем SSE и SSE2, благодаря их более высокой скорости и меньшему размеру генерируемого машинного кода.
- Рассмотренные выше способы не требуют вносить изменения в исходный код программ, поэтому обязательно должны использоваться в качестве первого (бесплатного) шага на пути оптимизации любых программ.
- В качестве дальнейших исследований, авторам представляется изучение особенностей более сложных способов оптимизации – например, ручного использования современных наборов SIMD (AVX, AVX2) в критических участках кода.

Список литературы

Intel Corporation. Intel(r) 64 and IA-32 Architectures Software Developers Manual. Number 325462-048US. September 2013.

Flops benchmark URL: https://github.com/Mysticial/Flops (дата обращения: 18.02.2019)

Jain, T. and Agrawal, T., 2013. The haswell microarchitecture-4th generation processor. International Journal of Computer Science and Information Technologies, 4(3), pp.477-480.

Pohl, A., Cosenza, B., Mesa, M.A., Chi, C.C. and Juurlink, B., 2016, March. An evaluation of current SIMD programming models for C++. In Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing (p. 3). ACM.

Бирюков Е.Д. Использование медианного фильтра в системе обработки изображений реалистичной компьютерной графики // Новые информационные технологии в автоматизированных системах: материалы семнадцатого научно-практического семинара - М.: ИПМ им. М.В.Келдыша, 2014, с. 216-220.