# Генерация кода для SSE/CUDA в задачах атмосферного моделирования

А. Н. Гаврилов

27 сентября 2009 г.(Draft)

Аннотация

Автоматизированная генерация кода из высокоуровневого описания позволяет легко адаптировать программу под существенно различные архитектуры, такие как Intel Core 2 (SSE2) и NVidia CUDA. Для реализации прототипа был использован язык Common Lisp в реализации ECL[1].

## 1 Свойства сеточных моделей

Наиболее часто используемым методом атмосферного моделирования являются сеточные алгоритмы. Они основаны на представлении состояния атмосферы в виде двумерных или трехмерных массивов. Размерности массива соответствуют реальным физическим измерениям с использованием постоянного шага, либо более сложной, но аналогичным образом фиксированной сетки.

Вычислительная часть модели состоит из набора формул, задающих взаимодействие между величинами. В ходе выполнения каждой итерации алгоритма они вычисляются для всех элементов выходного массива, получая на вход значения из соответствующих участков других массивов. Для крайних значений индексов часто используются специальные варианты формул. Сейчас эти формулы нередко выводятся с помощью систем вроде Maple[2]. Формулы могут содержать условные операторы, например для реализации ступенчатой функции.

Такая схема вычислений сама по себе очень хорошо поддается параллелизации. Однако реальные модели могут содержать следующие осложняющие факторы:

- Итерация может делаться не с единичным шагом, например, обращаться к элементам массива через один. Пропущенные индексы могут быть использованы при вычислении другой формулы, либо полностью игнорироваться.
- Оптимальное расположение размерностей массива в памяти, а также порядок циклов, может быть сложно определить с первого взгляда.
- Некоторые схемы сглаживания используют в формуле результат вычисления соседнего элемента вдоль одной из размерностей.

Хотя сами по себе эти проблемы обойти не сложно, ручная коррекция формул чревата возникновением трудно обнаружимых ошибок.

# 2 Целевые архитектуры

## 2.1 Intel Core 2 (SSE2)

Как и большинство современных процессоров общего назначения, Intel Core 2 – суперскалярная архитектура, способная выполнять в пределе до 4 команд за такт на каждом ядре. Все обращения к памяти идут через два уровня кэширования. На первом уровне каждое ядро имеет 32КБ со скоростью доступа порядка нескольких тактов. Второй уровень объемом до 12МБ используется сообща всеми ядрами.

Крупномасштабный параллелизм на данной архитектуре достигается путем разделения задачи на потоки по числу ядер. Каждый поток может выполнять любые команды, используя общую память для взаимодействия. С целью улучшения эффективности использования кеша, потоки имеет смысл привязывать к конкретным ядрам средствами операционной системы.

На низком уровне дополнительный параллелизм достигается путем использования команд, оперирующих 128-битными регистрами, каждый из которых может содержать 4 вещественных числа. Процессор предоставляет команды для обмена с памятью, арифметики[1], сравнений и побитных логических операций. Для их использования внутренний цикл разворачивается на 4 итерации; в связи с особенностями команд обмена с памятью для этого необходимо чтобы обращения к массивам шли с шагом 1.

Современные компиляторы способны автоматически использовать команды SSE в простых случаях, однако, как правило, не могут обработать условные операторы, хотя их возможно эмулировать с помощью команд сравнения и логических операций. Это связано с тем, что такая эмуляция требует вычисления обеих ветвей условия, что нарушает гарантии языка C.

Компиляторы также предоставляют возможность прямого использования векторных операций через специальные встроенные “функции”, транслируемые напрямую в соответствующие команды[3]. Компилятор при этом продолжает отвечать за распределение регистров и взаимодействие с памятью.

## 2.2 NVidia CUDA

В отличие от основного процессора, процессоры видеокарт NVidia[4] выполняют команды внутри одного потока строго в порядке их появления. Это компенсируется возможностью выполнения многих сотен потоков на одном ядре, с нулевой стоимостью переключения и аппаратным планировщиком.

Также отсутствует обязательное кеширование памяти. Вместо этого каждое ядро содержит 8 или 16 тысяч 32-битных регистров, распределяемых между выполняемыми потоками, а также 16КБ общей памяти для обме-

[1] Денормализованные числа замедляют вычисления в разы. Если задача позволяет, их можно отключить через управляющий регистр mxcsr.

на данными между ними[2]. Кроме этого, одновременное обращение многих потоков к соседним ячейкам памяти автоматически группируется в один запрос к внешней шине[3]. Наконец, в процессоре видеокарты есть кеш для доступа к текстурам, но их использование также требует явного изменения кода программы.

### 2.2.1 Блоки

На верхнем уровне параллелизации, задача разделяется на т.н. блоки, организованные в двумерную сетку. Размер сетки может задаваться во время исполнения, и ограничен только использованием для индексов 16-битных чисел. Размер сетки и координаты текущего блока доступны для использования в коде, исполняемом на видеокарте.

Блоки из сетки автоматически распределяются аппаратным планировщиком по свободным ядрам[4]. Каждое ядро может одновременно выполнять несколько блоков; их количество ограничивается потребностью блоков в регистрах и общей памяти, а также ресурсами планировщика[5].

Потоки внутри блока могут синхронизироваться с помощью специальной команды, а также обмениваться информацией через участок общей памяти. Сами блоки выполняются в произвольном порядке, и обмениваться информацией не могут.

### 2.2.2 Потоки

Каждый из блоков содержит одинаковое количество потоков, организованных в двумерную или трехмерную сетку. Общее количество потоков на одном ядре, а значит и в блоке, ограничено числом 512 или 1024[6] в зависимости от модели. Аналогично блокам, размер сетки задается во время исполнения, и доступен программе вместе с координатами потока. Все потоки выполяют одну и ту же процедуру, и изначально различаются только значениями своих координат.

Каждый поток получает собственный набор регистров из общего пула. Их распределение делается сразу по группам из 64 потоков. Практический опыт показывает, что для эффективного выполнения следует держать потребность потока в регистрах в пределах 64.

Аппаратный планировщик обрабатывает потоки статическими группами в 32 штуки (“warp”)[7]. В оптимальном случае он может одновременно выполнить одну и ту же команду для всех потоков группы[8]. В случае если потоки расходятся из-за выбора различных ветвей в условном операторе, планировщик вынужден выполнить обе ветви по очереди, временно

[2] При корректном использовании эта память работает с той же скоростью что и регистры, но требует дополнительного копирования данных.

[3] На эту оптимизацию сильно влияет выравнивание данных, особенно в старых моделях видеокарт.

[4] В зависимости от модели, процессор содержит от 2 до 30 ядер.

[5] Для эффективной работы необходима загрузка минимум 25%, т.е. 4 блока на ядро при 64 потоках на блок.

[6] В связи с пределом в 8 блоков, 100% загрузка требует минимум 128 потоков в блоке.

[7] Для того чтобы скрыть задержки на запись в регистры необходимо наличие минимум 6 активных групп (не обязательно в одном блоке).

[8] Реально ядро содержит только 8 вычислительных элементов и обрабатывает группу за 4 шага. Однако с точки зрения планировщика это атомарная операция.

отключая потоки, пошедшие по другой ветви. Это приводит к снижению производительности.

Группы, ожидающие данные из памяти, или момента синхронизации, временно блокируются. Остальные выполняются планировщиком по очереди. Кроме этого, в связи с тем, что для выполнения команды MADD используются специализированные элементы, она может выполняться параллельно с любой другой командой в другой группе.

### 2.2.3 Процесс разработки и исполнения

Для разработки процедур, выполняемых на видеокарте, используется язык C со специальными расширениями, и компилятор разработанный NVidia. Другие языки не поддерживаются.

Порожденный компилятором ассемблерный код загружается с помощью библиотеки, взаимодействующей с драйвером видеокарты. При этом он транслируется в машинный код соответствующей видеокарты[9]. Эта же библиотека отвечает за выделение памяти на видеокарте и копирование данных, а также позволяет запустить загруженные процедуры на исполнение, передав им нужные параметры.

# 3 Генерация кода

Наиболее удобным способом решения проблем с неоптимальными схемами индексирования массивов и итерации, а также адаптации программы к различным архитектурам, является генерация низкоуровневого кода из высокоуровневой спецификации. В случае если спецификация достаточно абстрактна, при изменении архитектуры достаточно изменить генератор кода. Кроме этого, схема, основанная на генерации, может помочь изолировать низкоуровневые параметры от основной логики программы.

## 3.1 Язык Лисп

Несмотря на 50-летнюю историю, язык Лисп до сих пор не потерял актуальность. В то время как многие его особенности, вроде сборки мусора и динамической типизации, были внедрены в новых популярных языках, он до сих пор лидирует по легкости своей модификации.

Это связано с изоляцией синтаксиса и семантики программы, а именно, использованием для записи кода простой структуры из вложенных списков. Такая организация позволяет легко выполнять произвольную трансформацию кода между синтаксическим анализом и компиляцией, что напрямую поддерживается стандартом языка в форме макросов.

В отличие от примитивных макросов языка C, макросы ANSI Common Lisp[5] являются полноценными функциями, которые отличаются от обычных только тем, что выполняются на этапе компиляции и манипулируют выражениями программы. Они также могут вызывать обычные функции,

[9] В этот момент происходит распределение реальных регистров. Ассемблер использует форму SSA.

в том числе из того же исходного файла, и использоваться сразу после объявления[10]. Значительная часть операторов из стандарта языка на самом деле реализуется с помощью макросов.

Язык также позволяет модифицировать работу самого синтаксического анализатора с помощью регистрации обработчиков для определенных комбинаций символов.

### 3.1.1 Преимущества для проекта

В связи с тем что Лисп – язык общего назначения, который при этом удобен для символьной обработки выражений, его можно использовать как для реализации генератора кода, так и для вычислительной программы. Это позволяет ограничиться одной средой разработки.

Кроме этого, за счет интеграции генератора кода в полноценный язык, его реализацию можно развивать постепенно, обращаясь к основному языку для выполнения редко используемых вычислительной программой, но необходимых операций.

Наконец, встроенная библиотека языка позволяет не заботиться о ручной реализации управления памятью, обработки ошибок и многопоточности.

### 3.1.2 Embeddable Common Lisp

Использованная в проекте реализация языка[1] как таковая заметно отстает от наиболее популярных свободно распространяемых реализаций как по производительности, так и по совместимости со сторонними библиотеками. Однако она легче всех интегрируется с кодом на C, что и является целью ее разработчиков.

В частности, поскольку при компиляции она транслирует код на язык C и вызывает внешний компилятор для получения загружаемой библиотеки, она позволяет вставлять в код на языке Лисп фрагменты программы на C, аналогично поддержке ассемблера компиляторами C и C++. Это позволяет генератору кода абсолютно прозрачным образом интегрировать результат своей работы в среду.

## 3.2 Расширения языка

В качестве средств высокоуровневой спецификации вычислительного алгоритма, язык был расширен следующими конструкциями.

### 3.2.1 Синтаксис ввода формул

Язык Лисп использует префиксный синтаксис, который неудобен для ввода и понимания больших формул. Кроме этого, формулы в такой записи нельзя получить средствами систем вроде Maple[2].

Для того чтобы обойти эту проблему, было реализовано синтаксическое расширение, распознающее формулы в инфиксной записи внутри фигурных скобок:

[10] Это отличается от систем вроде camlp4[6], где модификации синтаксиса компилируются отдельно и работают в среде, полностью отличной от основной программы.

```
{ NEW_DT[i,MW+1] := (TMP_ANU[MW-1]*TMP_EPS[MW+1]+TMP_ANU[MW+1])
                    / (1.0-TMP_EPS[MW+1]*TMP_EPS[MW-1]) }
```

Синтаксическое расширение распознает арифметические операции, обращения к массивам, вызовы функций, условия с операциями сравнения, и последовательности присваиваний.

### 3.2.2 Виртуальные массивы

Для оптимизации размещения массивов в памяти используется система виртуальных массивов. В момент их декларации задается набор логических размерностей, а также желаемый способ физической организации. Виртуальные массивы позволяют менять порядок физических индексов относительно логических, а также модифицировать структуру самих измерений одним из следующих способов:

1. Пропускать все элементы, кроме следующих с определенным шагом. Обращение к промежуточным индексам является ошибкой.

2. Добавлять дополнительное скрытое физическое измерение константного размера, и распределять элементы по полученной паре измерений, используя остаток от деления на эту константу. При этом итерация с шагом равным константе преобразуется в единичный шаг на физическом уровне.

Примеры деклараций:

```
(def-multivalue DR ((i 1 (1+ N1) :by 2) (k 1 (1+ MW) :by 2)))
(def-multivalue PL ((i 1 (1+ N1))       (k 1 (+ MW 2) :bands 2)))
```

Для обращения к виртуальным массивам используется макрос iref, аналогичный по внешнему интерфейсу оператору доступа к стандартным массивам aref. Система старается обнаруживать явные ошибки индексирования статически, и предупреждать о них на этапе компиляции.

Также предусмотрена возможность автоматической подстановки выражения вместо массива:

```
(def-multivalue-macro T_ (i k) { T0[k] })
```

### 3.2.3 Итерация по логическим размерностям

Для облегчения работы с виртуальными массивами предусмотрен оператор итерации по логическим индексам. Его параметрами являются имя виртуального массива и список его логических индексов. В результате раскрытия макросов генерируется ряд циклов по физическим размерностям массива, а вхождения имен логических индексов в тело цикла заменяются на соответствующие выражения.

```
(do-indexes OUT_U (i k)
  (format vel (formatter "~12,3E␣~12,3E␣~14,5E␣~14,5E~%")
          (iref xcoord i) (iref zcoord k)
          (iref OUT_U i k) (iref OUT_V i k)))
```

В случае если индексы используются для доступа к виртуальным массивам, при раскрытии iref эти выражения упрощаются до использования физических индексов напрямую.

Порядок индексов в заголовке цикла задает вложенность, а соответствие размерностям целевого массива определяется по совпадению имен индексов с идентификаторами в его декларации. Кроме этого, оператор позволяет задавать шаг и направление итерации, а также пропускать фиксированное количество элементов с обоих концов диапазона.

### 3.2.4 Оператор вычисления формулы

Наконец, система предоставляет оператор для вычисления массива по определенной формуле. Именно в нем происходит генерация оптимизированного кода. Внешне его заголовок похож на оператор цикла, но в данном случае порядок индексов должен соответствовать порядку соответствующих измерений, а вложенность циклов определяется автоматически. В примере можно увидеть как задается шаг цикла[11] и пропуск элементов:

```
(compute PL ((i :skip (1 1) :step (* 2))
             (k :step (* 2)))
  { t3*t10/2.+2.*t1*_grp(t2/t3)*t12 }
  :parallel i
  :with { t1 := PL[i+1,k];
          ... })
```

С помощью дополнительных именованных параметров можно объявлять временные переменные для использования в основной формуле, задавать зависимость между итерациями по определенному индексу, контролировать распараллеливание в режиме SSE и задавать дополнительные флаги для CUDA. Путем использования последовательности присваиваний вместо основного выражения можно одновременно обновлять несколько массивов, а указав в заголовке константу или арифметическое выражение вместо индекса можно избежать генерации соответствующего цикла.

В случае невозможности транслировать выражение для SSE или CUDA, оператор раскрывается в эквивалентный цикл на языке Лисп. Это позволяет использовать тот же синтаксис и для операций, не поддержанных генератором кода:

```
(compute AMU (k) (call-fun molecular-mass (iref zcoord k)))
```

# 4 Оптимизация

Для улучшения производительности система использует определенный набор техник оптимизации. Часть из них направлена на реструктуризацию формул, часть является специфичной для адаптации их к выполнению на процессоре видеокарты.

## 4.1 Реструктуризация выражений

Помимо арифметического упрощения индексных выражений, необходимого для эффективной реализации перехода от логических к физическим индексам и наоборот, программа выполняет ряд шагов, направленных на улучшение эффективности вычислений:

[11] Запись “(* число)” означает, что задается шаг, но не направление.

1. Деревья из вложенных ассоциативно-коммутативных операций (сложения и умножения) уплощаются, а их операнды сортируются для выявления реальной структуры выражений. В ситуациях, когда это недопустимо, может быть использован специальный оператор _grp(...)[12], действующий как не подлежащие разбиению скобки.

2. Выполняется простейшее вынесение общих сомножителей за скобки, сокращение противоположных слагаемых и множителей.

3. Выполняется упрощение выражений, используемых для ветвления по знаку[13], за счет использования предоставленной программистом информации о знаке значений, хранящихся в переменных и массивах.

4. Уплощенные группы слагаемых и множителей разбиваются с учетом их зависимости от индексов разного уровня вложенности циклов. Это улучшает эффективность вынесения из циклов инвариантных выражений.

5. Группы слагаемых и множителей дополнительно разбиваются с целью повышения эффективности выделения общих подвыражений. Для этого выполняется их попарное сравнение и выделение обнаруженных таким образом общих подмножеств. При работе используется очередь подмножеств, упорядоченная по убыванию их размера.

6. Выполняется выделение общих подвыражений с распределением кода по уровням вложенности циклов.

7. Уплощенная структура преобразуется в бинарное дерево с учетом необходимости балансировки и минимизации числа операций деления.

## 4.2 Особенности CUDA

Особенности архитектуры графического процессора требуют применения дополнительных шагов оптимизации. Их можно разделить на меры по улучшению эффективности использования памяти, и реорганизации структуры циклов.

### 4.2.1 Эффективное использование памяти

Графический процессор содержит целый ряд различных видов памяти и способов доступа к ней. Их эффективное использование необходимо для получения хорошей производительности. В частности, генератор кода:

- Поддерживает использование механизма текстур для чтения из массивов.

  Текстуры предоставляют возможность кеширования данных, умешьная нагрузку на интерфейс с внешней памятью. В связи с неясностью

[12] В отсутствие такой конструкции, перестановка операндов может в итоге понизить полезность системы т.к. она в некоторых случаях недопустима из-за округления и ограничений по диапазону вещественных чисел.

[13] В модели атмосферы, для оптимизации которой был разработан данный генератор кода, часто используется ступенчатая функция (ifsign x 0.0 0.5 1.0). В ее аргументах нередко содержатся константы и другие сомножители, не влияющие на знак.

границ эффективности данной оптимизации, ее использование необходимо задавать вручную. Однако генерация кода позволяет упростить это до простого указания списка имен массивов, к которым ее нужно применить.

- Автоматически выявляет возможность размещения временного массива в регистрах.

  Временные массивы используются для реализации переноса значений между итерациями, а также выделения параллелизуемых частей циклов с таким переносом. В случае когда каждый конкретный элемент массива используется только в одном потоке, массив можно разместить в локальных регистрах. Это позволяет уменьшить расход общей памяти и избежать необходимости синхронизации.

- Автоматически размещает постоянные значения в общей памяти.

  Значения, одинаковые для всех потоков блока, могут быть размещены в общей памяти. Предельным случаем такого размещения являются параметры, которые задаются один раз при запуске процедуры, и просто копируются планировщиком в каждый блок. Значения, которые зависят от индекса блока, могут быть посчитаны одним из потоков в начале его работы. Это преобразование позволяет уменьшить число необходимых регистров.

### 4.2.2 Разбиение направленных циклов

В норме самый внутренний цикл распределяется генератором кода по потокам внутри блока и выполняется параллельно. Однако если задача требует передачи значения между итерациями, они вынужденно выполняются последовательно.

В связи с особенностями организации вычислений в процессоре, такое упорядочивание повышает стоимость каждой команды в теле цикла более чем на порядок. Таким образом, в отличие от случая вычислений на центральном процессоре, эффективность требует вынесения за пределы цикла всех без исключения операций, которые можно выполнить параллельно.

Возникновение при этом временных переменных не представляет значительной проблемы, т.к. все они размещаются в локальных регистрах потоков.

Для выполнения данной оптимизации генератор кода выявляет переменные, которые одновременно читаются и пишутся в цикле, и оставляет в упорядоченной части только действия, которые участвуют в цепочке зависимостей.

### 4.2.3 Перестановка операций в теле цикла

Опыт показывает, что перестановка операций в теле самого внутреннего цикла способна заметно понизить потребность кода в регистрах, что важно для достижения оптимальной производительности.

Для выполнения перестановки все выражения разбиваются на элементарные операции и конвертируются в последовательность присваиваний временным переменным. После этого присваивания перераспределяются с использованием следующих эвристик:

1. Для упорядочивания используется очередь из операций, все операнды которых уже посчитаны.
2. На каждом шаге выбирается операция с максимальным числом операндов.
3. Если таких в очереди несколько, выбирается идущая первой в исходном порядке[14].

Эффективность таких простых эвристик может свидетельствовать о том, что механизм распределения регистров в компиляторе NVidia использует алгоритм, не поддерживающий возможность существенной реорганизации последовательности операций. Кроме этого, в некоторых случаях число реально использованных регистров зависит от заданного ограничения на их количество.

# 5 Результаты

Генератор кода был использован для оптимизации работы реализации двумерной модели атмосферы [7].

- Режим SSE дал ускорение в 14 раз относительно исходной реализации на языке Фортран с использованием OpenMP.
  1. Ускорение в 4 раза является ожидаемым результатом использования векторных команд.
  2. Упрощение выражений дало выигрыш порядка 40%
  3. Оставшиеся 2.5 раза, вероятно, можно объяснить более эффективным порядком обхода памяти, удалением лишних вызовов функций, и.т.п.
- Режим CUDA на видеокарте GeForce GTX 275 дал дополнительное ускорение до 5 раз относительно варианта SSE на процессоре Intel Core 2 Quad 2.66 GHz, запущенного в 3 потока. Точное значение ускорения растет по мере увеличения размера массивов.

Сравнение спецификаций производительности обоих процессоров позволяют ожидать предельное ускорение порядка 26 раз. Однако в случае невозможности использования команд MADD на видеокарте данный показатель падает до числа 8. Кроме того, грубая оценка показывает, что зависимости между итерациями внутреннего цикла в используемой данной моделью вычислительной схеме может легко привести к замедлению в 1.5–2 раза.

# Список литературы

[1] Реализация Embeddable Common Lisp, http://ecls.sourceforge.net/

[14] Данный порядок соответствует двухуровневому обходу в глубину слева направо: сначала по общим подвыражениям, а внутри по всем операциям.

[2] Система компьютерной алгебры Maple,
http://www.maplesoft.com/
http://en.wikipedia.org/wiki/Maple_%28software%29

[3] Встроенные функции MMX, SSE и SSE2
http://msdn.microsoft.com/en-us/library/y0dh78ez%28VS.80%29.aspx

[4] Документация и средства разработки CUDA.
http://www.nvidia.com/object/cuda_learn.html

[5] Спецификация ANSI Common Lisp,
http://www.lispworks.com/documentation/common-lisp.html

[6] Препроцессор CamlP4,
http://caml.inria.fr/pub/docs/manual-camlp4/index.html

[7] Kshevetskii S P, Gavrilov N M. Vertical propagation, breaking and effects of nonlinear gravity waves in the atmosphere. J Atmos Sol Terr Phys, 2005, 67, 1014—1030