Анализ эффективности решения задачи N тел на различных вычислительных архитектурах

А.В. Адинец

Задача N тел представляет собой расчет поведения системы тел, взаимодействующих при помощи дальнодействующих сил. Различные варианты задачи используются в астрономии и молекулярной динамике, где рассматриваются тела, взаимодействующие при помощи гравитационных и межмолекулярных сил соответственно. Данная задача характеризуется очень высокой вычислительной сложностью: для расчета взаимодействия N тел требуется выполнить $O(N^2)$ операций. Как следствие, для решения задачи требуется использовать параллельные вычисления и суперкомпьютеры. В данной работе рассматриваются реализации решения задачи N тел на различных вычислительных архитектурах: графических процессорах AMD и NVidia, процессорах CELL, многоядерных процессорах, кластерных системах. Для реализации используются технологии C$, SPU-C++, SSE, OpenMP, MPI, а также их комбинации. Для различных архитектур наблюдается различие производительности тривиальной и оптимизированной версий в 5 – 10 раз. Также показывается, что повышение производительности на разных архитектурах достигается использованием одних и тех же или сходных оптимизаций.

1. Введение

Задача N тел является задачей расчета эволюции поведения системы из N тел, которые взаимодействуют при помощи некоторых сил [1]. Эта задача возникает при численном моделировании процессов в различных областях науки: молекулярной динамике, астрономии, жидкостной динамике, электростатике. При этом силы, действующие между частицами в модели, как правило, имеют дальнодействующий характер. Если количество частиц N, то количество взаимодействий между ними, которые необходимо учитывать, растет как $O(N^2)$. Это обуславливает высокую вычислительную сложность задачи.

Помимо тривиального алгоритма взаимодействия, существуют различные приближенные схемы [2]. В среднем они позволяют снизить сложность с $O(N^2)$ до $O(N \log N)$ или даже $O(N)$, однако имеют худшие показатели точности. Не для всех имеются оценки ошибки сверху. Более того, для расчета взаимодействия в скоплениях близко расположенных частиц часть этих методов все равно использует взаимодействие типа каждый-с-каждым внутри скопления. А поскольку общее число частиц может быть достаточно большим, размер среднего скопления также оказывается большим. Что также требует много вычислительных ресурсов для расчета $O(N^2)$ взаимодействий, где N в этом случае является размером скопления.

Таким образом, для решения задачи N тел в любом случае требуются большие вычислительные ресурсы. В настоящее время, существует целый ряд высокопроизводительных вычислительных архитектур. Прежде всего, это традиционные центральные процессоры, а также построенные на их основе вычислительные кластеры. Далее идут системы на основе процессоров CELL BE [3], а также графические процессоры [4]. Наконец, для решения задачи N тел, взаимодействующих по гравитационному закону Ньютона, созданы специальные аппаратные решения, к примеру, семейство систем GRAPE [5].

Данная статья рассматривает реализацию наиболее ресурсоемкой части задачи N тел, расчета взаимодействия частиц «каждый-с-каждым», на различных современных вычислительных архитектурах. Для каждой архитектуры рассматривается несколько реализаций, от самой простой к более эффективной, с указанием используемых оптимизаций. Анализируются факторы, влияющие на производительность.

Задача N тел позволяет также оценить эффективность компилятора для данной архитектуры на вычислительно емких, но нетривиальных задачах. Для различных архитектур проводится сравнение эффективности решения на них данной задачи.

Данная статья организована следующим образом. В разделе 2 дается математическая формулировка задачи N тел дается краткий обзор методов ее решения. В разделе 3 приводится описание и сравнение современных вычислительных архитектур: графических процессоров, процессора CELL, а также обычных процессоров и кластерных систем на их основе. В разделе 4 для каждой из архитектур рассматриваются особенности реализации вычислительного ядра задачи N тел и оптимизации, применяемые программистом или компилятором. В разделе 5 приводятся результаты вычислительных экспериментов. При этом сравниваются оптимизированные и неоптимизированные версии программы для одной и той же архитектуры, обсуждается эффективность решения данной задачи. В разделе 6 кратко суммируются основные результаты статьи.

2. Задача N тел.

В задаче N тел взаимодействие вычисляется отдельно между парами частиц, а сила, действующая на каждую частицу (*i-частицу*) является суммой вкладов отдельных частиц (*j-частиц*), которые на нее действуют. Каждая частица характеризуется набором параметров, таких как масса или заряд. Кроме того, каждая частица характеризуется скоростью $\mathbf{v}_i$ и положением $\mathbf{r}_i$. Задачей *ядра вычисления взаимодействия* является расчет ускорения, которые частица получает в результате влияния на нее других частиц:

$$\mathbf{a}_i = \frac{1}{m_i} \sum_{j=1, j\neq i}^{N} \mathbf{f}_{ij} \ , \qquad (1)$$

где $\mathbf{f}_{ij}$ - сила, с которой j-частица действует на i-частицу. При этом сила взаимодействия двух частиц вычисляется по некоторому закону и зависит, вообще говоря, от их взаимного положения, скорости, а также характеристик частиц. Примеры потенциалов взаимодействия приведены в (1). Для гравитационного взаимодействия, на основании формулы (1), имеем:

$$\mathbf{a}_i = G \sum_{j=1, j\neq i}^{N} \frac{m_j}{\left|\mathbf{r}_j - \mathbf{r}_i\right|^3} (\mathbf{r}_j - \mathbf{r}_i) . \qquad (2)$$

Непосредственные вычисления по формуле (2) являются достаточно ресурсоемкими: для N частиц объем вычислений растет как $O(N^2)$. Поэтому помимо прямых методов, для эффективного моделирования используются и более сложные схемы, позволяющие сократить объем вычислений.

Прямые схемы [6] для интегрирования обычно используют схему Эрмита, которая требует вычисления не только ускорения, но и его производной по времени. Для каждой частицы поддерживается собственное значение времени и шага по времени, которое округляется до ближайшего значения на дискретной сетке, обычно экспоненциальной. В качестве шага по времени берется минимальный шаг из всех частиц. При этом положение и скорость обновляются только для тех частиц, для которых настало время, т.е. текущее время равно их шагу по времени плюс время частицы.

Древовидные схемы [7] позволяют сократить сложность вычислений до O(N log N). В этом случае влияние дальних групп тел вычисляется приближенно как взаимодействие с центром масс. Точность такой схемы оказывается ниже, и для некоторых реальных астрономических конфигураций ее использование может привести к неограниченным ошибкам в симуляции. Более того, при использовании данной схемы требуется вычислять взаимодействие между группами частиц, число которых может быть достаточно большим.

Мультипольные схемы [2] обладают ограниченной масштабируемостью при реализации на параллельных архитектурах. Сеточные методы, решающие уравнение Пуассона на сетке при помощи преобразования Фурье, требуют использования адаптивных сеток для достижения эффективности.

Анализ эффективности решения задачи N тел ранее проводился на различных вычислительных архитектурах. Рассматривались традиционные процессоры [8], кластеры на их основе [9] и графические процессоры [10], [11]. Публикации по использованию процессора CELL BE для решения гравитационной задачи N тел авторам статьи неизвестны. Однако кластер Roa-

drunner на базе процессоров CELL использовался для моделирования межмолекулярного взаимодействия с короткодействующим потенциалом [12]. Помимо этого, авторам неизвестны работы, где проводилось бы сравнение реализации задачи N тел на достаточно большом количестве архитектур.

3. Целевые архитектуры и средства программирования.

3.1. Графические процессоры.

Современные графические процессорные устройства (ГПУ) [4] в настоящее время активно используются для решения ресурсоемких задач. Соответствующее направление получило название *общих вычислений на ГПУ* (ОВГПУ, калька с англ. GPGPU – General Purpose GPU). Основными причинами этого является их высокая производительность [13], [14] и высокая доступность, а также лучшие показатели энергетической эффективности и стоимости по сравнению с традиционными архитектурами.

В настоящее время существуют 2 основных производителя ГПУ для общих вычислений: компании AMD и NVidia. ГПУ этих производителей, хотя и отличаются в деталях, и используют различные технологии программирования, имеют сходную архитектуру. Общая схема архитектуры современных ГПУ и их места в *видеокарте,* плате, приведена на рисунке 1.

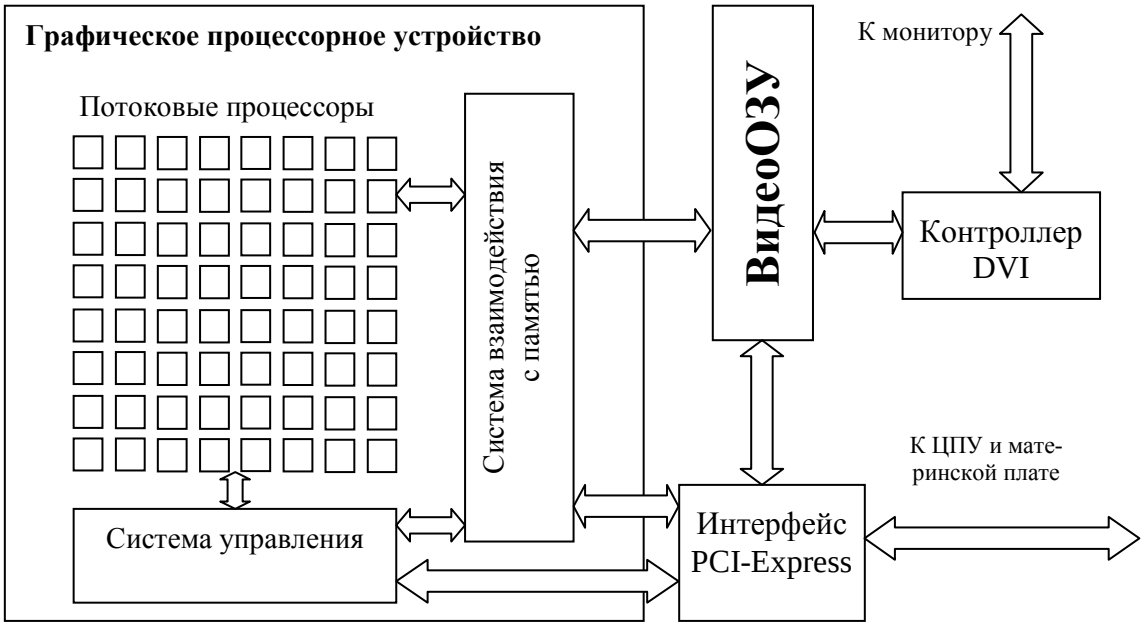

Рисунок 1. Общая схема современного ГПУ и его места в видеокарте и ПК.

Основными вычислительными устройствами ГПУ являются *потоковые процессоры (ПП)*. Их система команд включает арифметические операции с целыми и вещественными числами, а также команды асинхронного доступа к *видеоОЗУ*, ОЗУ ГПУ. При вычислениях на ГПУ все ПП исполняют одну и ту же вычислительную программу, называемую *шейдером*. Программа выполняется для всех узлов некоторой целочисленной решетки, размер которой может намного превосходить число ПП на ГПУ. Один запуск шейдера на такой целочисленной решетке называется *проходом*. Проход запускается под управлением *системы управления* ГПУ, которая отвечает за загрузку программы, распределение точек целочисленной решетки между ПП и т.д. На одних ГПУ все ПП имеют один и тот же счетчик команд, на других ГПУ они объединены в группы, имеющие общий счетчик команд. Таким образом, современные ГПУ представляют собой пример архитектуры типа ОКМД с общей памятью.

Проблема дисбаланса производительности процессора и памяти на ГПУ решается при помощи сокрытия задержек с использованием *аппаратной многопоточности*. *Поток ГПУ* – это

исполнение шейдера для одной точки целочисленной решетки. Если один из потоков заблокировался на доступ к памяти, *система управления* передает исполнение на другой поток. Поскольку контекст потока состоит только из регистров, переключение потоков осуществляется быстро.

Различия между архитектурами ГПУ различных производителей можно посмотреть, к примеру, в [15]. Там же приведен обзор средств программирования ГПУ. В качестве средства реализации на ГПУ задачи N тел выбрана система C$ [16]. Она позволяет иметь общую базу кода, для различных ГПУ, кроме того, может автоматически оптимизировать код под конкретный ГПУ. Более того, одной из целей статьи является тестирование компиляторов для различных вычислительных платформ, в том числе и компилятора C$.

3.2. Процессор CELL BE.

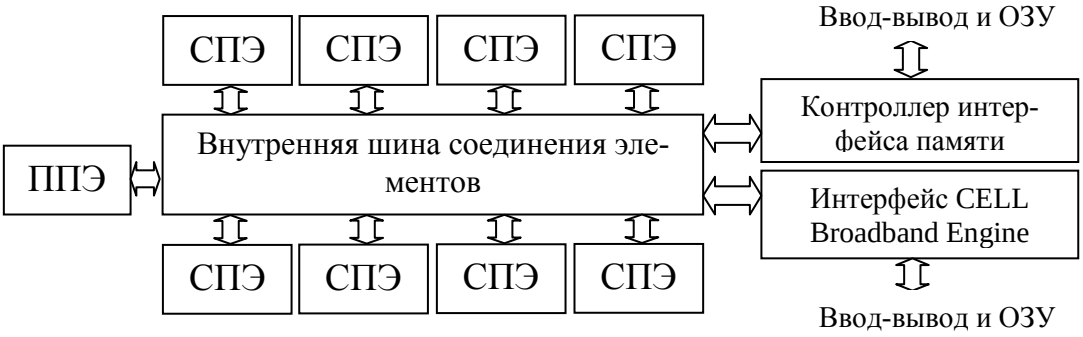

Рисунок 2. Общая архитектура процессора CELL BE.

Процессор CELL BE занимает промежуточное место между традиционными многоядерными процессорами и ГПУ. Общая схема его архитектуры приведена на рисунке 2. Сам процессор состоит из набора главных и вспомогательных ядер. Главные ядра в основном выполняют управляющую функцию, хотя могут использоваться и для вычислений. Вспомогательные ядра выполняют только вычисления.

В роли главных ядер выступает PowerPC Processing Element (PPE, ППЭ – калька с английского). В роли вычислительных ядер выступают синергические процессорные элементы (СПЭ, англ. Synergistic Processing Element, SPE). Каждый СПЭ состоит из синергического процессорного устройства (СПУ, англ. Synergistic Processing Unit, SPU) и контроллера потока памяти (КПП, англ. Memory Flow Controller, MFC). СПУ представляет собой векторный RISC-процессор, имеющий 128 регистров, каждый из которых имеет размер 128 бит. Каждый СПУ также имеет в своем распоряжении 256 КБ явно адресуемой быстрой локальной памяти (ЛП).

СПУ не может напрямую обращаться к ОЗУ, это выполняется при помощи КПП. КПП является вспомогательным процессором и может выполнять асинхронную передачу данных из ОЗУ объемами до 16 КБ. Все элементы процессора CELL соединены внутренней шиной соединения элементов (англ. Element Interconnection Bus, EIB).

Для программирования CELL используется язык C/C++ с расширениями и библиотекой программирования, входящими в IBM CELL SDK (3). В настоящее время необходимые расширения поддерживаются компиляторами gcc и IBM XLC/C++. Запуск программ на СПЭ осуществляется при помощи создания контекста СПЭ с загруженной в него программой, создания отдельного POSIX-потока и последующим исполнением в нем контекста СПЭ.

3.3. Обычные центральные процессоры и кластерные системы.

В качестве «обычных» процессоров выступили четырехъядерные процессоры Intel Xeon E5472 (Harpertown) [17]. Из архитектурных особенностей следует отметить многоядерность, поддержку команд Streaming SIMD Extension (SSE) [18], а также кэш-память 1-ого и 2-ого уровней.

Для программирования использовался язык C++ и компилятор Intel C++. Многоядерность процессора задействовалась при помощи использования технологий OpenMP при работе на од-

ном узле и MPI при работе на кластере. Для достижения максимальной производительности технология SSE задействовалась при помощи использования встроенных функций (англ. intrinsics) и специальных типов данных.

3.4. Сравнение различных архитектур.

Таблица 1. Сравнение характеристик тестовых систем.

Архитектура	ГПУ AMD HD 4850	ГПУ NVidia 8800GT	CELL BE	Intel Xeon E5472
Пиковая производительность, ГФлопс (одинарная/двойная точность)	800 / 160	340 / --	204 / 102	96 / 48
Количество ядер	160 (ПП)	128 (ПП)	8 (СПЭ)	4 (x86)
Регистров (на ядро)	до 128 x float4	до 64 x float	128 x float4	16 x float4 + обычные
Система команд ядра	VLIW/векторные трехадресные RISC	скалярные трехадресные RISC	векторные трехадресные RISC	векторные + скалярные двухадресные CISC
Память на чипе	L1: ~ 16 кбайт	16 кбайт явно адресуемой памяти на блок потоков	256 кбайт явно адресуемой памяти на СПУ	L1: 64 кбайт данные + 64 ki кода L2: 12 Мбайт
Пропускная способность канала процессор – ОЗУ	108 Гбайт/с	70 Гбайт/с	25 Гбайт/с	10 Гбайт/с
Доступ к ОЗУ	кэшируемый асинхронный	некэшируемый асинхронный	некэшируемый асинхронный	кэшируемый синхронный
Место в вычислительной системе	подчиненный (через PCI-E)	подчиненный (через PCI-E)	главный или подчиненный	главный
Мощность процессора/системы, Вт	160 / 500	145 / 400	- / 217	80 / 416[1]
Энергетическая эффективность процессора/системы, МФлопс/Вт	5000 / 1600	2340 / 850	- / 1840[2]	600 / 230

Сравнение характеристик различных архитектур приведено в таблице 1. ГПУ лидируют по показателям абсолютной производительности и энергетической эффективности процессора. Однако они отстают от процессора CELL по показателю энергетической эффективности системы в целом.

Также можно заметить сходство различных вычислительных архитектур. Все архитектуры, за исключением ГПУ NVidia, поддерживают векторные команды. В более новых процессорах используются трехадресные команды, в более старых – двухадресные. Для работы с векторными операндами используются регистры, содержащие четверки вещественных чисел. Заметим, что максимальная производительность на этих архитектурах может быть достигнута только за счет использования векторных команд.

Следует обратить внимание на иерархию памяти каждой из архитектур. ГПУ AMD и x64 используют кэш-память, в то время как в случае ГПУ NVidia и процессора CELL память на чипе является явно адресуемой. В процессоре CELL использование памяти на чипе обязательно,

[1] В качестве системы рассматривается кластер «СКИФ-Чебышев», в дальнейшем используемый в вычислительных экспериментах

[2] В качестве системы рассматривается IBM QS22 Blade Server

поскольку процессор не может получать доступ к данным ОЗУ напрямую. На ГПУ NVidia это необязательно, при этом программа, не использующая явно адресуемую память на чипе, будет работать медленно.

Все рассматриваемые архитектуры предоставляют возможность асинхронного доступа к ОЗУ. В случае ГПУ команды доступа к памяти являются асинхронными. В случае процессора CELL асинхронный доступ к ОЗУ осуществляется при помощи КПП, а в случае процессора x64 – при помощи команд предвыборки данных в кэш. Учет характера иерархии памяти в программе и грамотное использование асинхронного доступа позволяет повысить эффективность программы.

4. Особенности реализации алгоритма N тел для различных вычислительных архитектур.

В данном разделе рассказывается об особенностях реализации алгоритма расчета ускорения по формуле (2). При этом расчет ускорения ведется в одинарной точности, поскольку расчеты с двойной точностью реализованы эффективно не на всех архитектурах.

4.1. Реализация для графических процессоров.

Реализация для ГПУ выполнена с использованием языка C$. С одной стороны, он позволяет создавать высокоуровневые программы, которые будут эффективно исполняться на ГПУ различных производителей [16]. С другой стороны, язык требует записи программы в высокоуровневых терминах, так что программист лишен возможности выполнять низкоуровневые оптимизации, которые ему доступны, к примеру, в системе CUDA [19].

Фрагмент реализации задачи N тел на ГПУ на языке C$ приведен на рисунке 3.

```
// computes pairwise acceleration
public float3 accel(float mj, float3 ri, float3 rj) {
  float3 dr = rj - ri;
  float len2 = dr.len2;
  float3 a = mj / (len2 * sqrt(len2) + eps2) * dr;
  return a;
}  // end of accel()
...
// computes acceleration for all particles
public void float3[] accelGpu() {
  a = let(i) G * + accel(ms[j], fr[i], fr[j]);
}  // end of accelGpu()
```

Рисунок 3. Фрагмент реализации задачи N тел на языке C$.

Поскольку программист в системе C$ не имеет прямого контроля над низкоуровневыми оптимизациями, программа в системе запускалась один раз с включенными и один раз с выключенными оптимизациями. При включенных оптимизациях используется развертка редукций и векторизация. Для различных ГПУ задействовалась их иерархия памяти: для ГПУ AMD использовался большой регистровый файл, для ГПУ NVidia – статическая разделяемая память.

4.2. Реализация для обычных процессоров, кластерных систем и CELL.

Для обычных процессоров система создано приложение командной строки, реализующее расчеты по задаче N тел с использованием схемы Эйлера для интегрирования. Тип используемых оптимизаций, количество частиц и прочие настройки передаются как параметры командной строки.

Рассматриваются следующие оптимизации:

- Использование технологии SSE
- Использование нескольких ядер при помощи технологии OpenMP
- Использование нескольких узлов кластера при помощи технологии MPI

Во избежание дублирования кода, обычные и SSE версии задачи N тел для обычных процессоров реализованы при помощи общего шаблона на C++.

Основным объектом программы является *решатель* (Solver) задачи N тел. Он выполняет загрузку и выгрузку данных из рабочего набора, а также последовательный вызов различных шагов процедуры решения.

За представление данных во время выполнения расчета отвечает класс *рабочего набора*. Он определяет используемые для представления данных типы и размещение данных в оперативной памяти. За обработку данных отвечают классы-*интерактор*ы, задачей которых является обеспечение расчета взаимодействия i-частиц с j-частицами. Каждый интерактор может работать только с одним типом рабочего набора. С другой стороны, с каждым рабочим набором может быть связано несколько типов интеракторов. Например, с обычным локальным рабочим набором может быть связан как одноядерный интерактор, так и многоядерный интерактор, использующий технологию OpenMP. Рабочие наборы и интеракторы могут быль *локальными* и *распределенными*. Первые отвечают за вычисления на одном узле, вторые – за объединение узлов в кластер и взаимодействие между узлами.

Явное использование расширения SSE реализовано при помощи использования встроенных функций и типов (intrinsic). Чтобы ограничить зависимость кода от низкоуровневых особенностей архитектуры, работа с четверками вещественных чисел инкапсулирована в классе qfloat. Для него реализован набор стандартных арифметических операций и элементарных функций. Используемая в архитектуре схема рабочих наборов и интеракторов позволяет иметь один и тот же код для работы как с векторными командами, так и с обычными. Более того, такая схема позволяет легко инкапсулировать и использовать новые особенности архитектуры, например, 8- или 16-элементные векторные операции, которые используются в процессорах Intel Nehalem и Intel Larrabee, соответственно.

При описанном выше подходе большая часть кода, специфичная для процессора CELL, уже присутствовала в готовом виде после выполнения реализации для процессоров x64.

Для СПУ потребовалось реализовать только отдельный интерактор и код, работающий на СПУ. Специальный код для CELL потребовался ввиду особенностей работы с памятью CELL. Помимо этого, один из внутренних циклов потребовалось развернуть вручную.

5. Результаты вычислительных экспериментов.

5.1. Параметры эксперимента и тестовые системы.

Основной задачей вычислительных экспериментов являлось определение эффективности реализации задачи N тел на различных вычислительных архитектурах, а также факторов, влияющих на эффективность. Для каждого значения архитектуры рассматриваются разные количества взаимодействующих частиц.

В литературе по задаче N тел принято считать, что количество вещественных операций при вычислении одного взаимодействия составляет 38, при этом дается ссылка на статью [20], где впервые появляется эта цифра. При этом в самой статье указывается, что 38 – это число, которое получается при использовании конкретной схемы аппроксимации величины обратного квадратного корня на данной архитектуре; для других архитектур цифра может быть другой. На графических процессорах, к примеру, больше подойдет цифра 24, поскольку вычисление квадратного корня или обратной величины там имеет стоимость, равную стоимости 4-х обычных операций. Для нашей текущей реализации на CELL подходящая цифра была бы 32. В данной работе для оценки производительности считается, что вычисление одного взаимодействия занимает 24 операции, вне зависимости от используемой архитектуры.

В качестве тестовых ГПУ использовались AMD HD Radeon 4850 с пиковой производительностью 800 ГФлопс и NVidia GeForce 8800GTX с пиковой производительностью 340 ГФлопс. В качестве системы на базе процессора CELL BE выступал IBM QS22 Blade Server, содержащий 2 процессора CELL в качестве основных, всего 16 СПУ. Код для процессоров x64 тестировался на кластере «СКИФ-Чебышев», узлы которого содержат по 2 4-ядерных процессора Intel Xeon E5472 с пиковой производительностью 96 ГФлопс (с одинарной точностью). В зависимости от параметров, программа запускалась на отдельном ядре, узле или множестве узлов.

5.2. Вычислительные эксперименты на ГПУ.

Таблица 2. Результаты вычислительных экспериментов на ГПУ.

ГПУ	Число частиц	Оптими-зации	Производитель-ность, ГФлопс	Эффективность
NVidia GeForce 8800GTX	4096	Нет	9,17	2,70%
NVidia GeForce 8800GTX	32768	Нет	10,58	3,11%
NVidia GeForce 8800GTX	32768	Да	209,93	61,74%
NVidia GeForce 8800GTX	110592	Да	211,60	62,24%
AMD Radeon HD4850	4096	Нет	113,24	14,15%
AMD Radeon HD4850	32768	Нет	127,71	15,96%
AMD Radeon HD4850	32768	Да	678,19	84,77%
AMD Radeon HD4850	110592	Да	716,53	89,56%

Результаты тестов на ГПУ даны в таблице 2. Эффективность, достигаемая на ГПУ AMD и NVidia при решении задачи N тел, позволяет говорить об эффективности используемых оптимизаций. Для ГПУ NVidia производительность C$-программы близка к производительности программы, полученной в результате ручной оптимизации, которая составляет от 200-240 ГФлопс.

В качестве оптимизаций для ГПУ AMD система выполняет векторизацию цикла по i-частицам и развертку цикла суммирования взаимодействий по j-частицам. Глубина развертки внутреннего цикла составляет 16. Команды обращения в видеоОЗУ при этом переносятся в самое начало цикла. Для ГПУ NVidia система выполняет развертку цикла по j-частицам, а также задействует явно адресуемую память на чипе.

Включение оптимизаций C$ на обоих типах ГПУ дает повышение производительности в 5-10 раз по сравнению с неоптимизированным вариантом. Объясняется это тем, что C$ - достаточно высокоуровневый язык, который не дает пользователю доступа к низкоуровневым особенностям ГПУ. Поэтому их должна использовать система. Высокая достигаемая производительность свидетельствует, с другой стороны, об эффективности используемых оптимизаций для различных архитектур.

5.3. Вычислительные эксперименты на процессорах CELL.

Таблица 3. Производительность решения задачи N тел на процессоре CELL.

Число частиц	Векторизация	Производительность, ГФлопс	Эффективность
13824	Да	3,86	0,95%
13824	Нет	75,38	18,48%
32768	Да	3,90	0,96%
32768	Нет	102,64	25,16%
110592	Да	3,91	0,96%
110592	Нет	109,13	26,75%

Результаты вычислительных экспериментов на процессоре CELL приведены в таблице 3. Для каждого значения числа частиц указана производительность кода с векторизацией и без нее, на 2-х процессорах CELL (16 СПЭ). Векторизация дает значительный прирост производительности по сравнению с использованием обычного кода. Для большого числа частиц может достигаться более чем 20-кратное ускорение. Такое большое значение ускорения объясняется тем, что система команд СПЭ поддерживает *только векторные операции*. Таким образом, использование скалярных операций влечет за собой высокие накладные расходы, которые не возникают в коде, который использует векторизацию.

В случае использования векторизации задача является ограниченной производительностью АЛУ СПЭ, а не обменом данными с памятью. Использование двойной буферизации и другие

оптимизации работы с памятью, обычно рекомендуемые для процессора CELL [3], не дают значительного роста производительности.

Данная задача практически идеально масштабируется по числу СПЭ. Этот результат вполне ожидаемый, поскольку на каждой итерации элементы целевого массива данных обрабатываются независимо.

5.4. Вычислительные эксперименты на обычных ЦПУ.

На обычных ЦПУ отдельно проводились эксперименты на одном узле и на кластере. Результаты экспериментов на одном узле на одном ядре процессора приведены в таблице 4, на всем узле с использованием OpenMP – в таблице 5. В таблице 4 эффективность дается в расчете на одно ядро, в таблице 5 – на весь узел.

Таблица 4. Результаты вычислительных экспериментов на x64 на одном ядре.

Число частиц	Векторизация	Производительность	Эффективность
27000	Нет	3,25	13,56%
27000	Да	11,75	48,97%
64000	Нет	3,25	13,56%
64000	Да	11,75	49,00%

Таблица 5. Результаты вычислительных экспериментов на одном узле (8 ядер).

Число частиц	Векторизация	Производительность	Эффективность
64000	Нет	26,16386	13,63%
64000	Да	94,77419	49,36%
125000	Нет	26,15014	13,62%
125000	Да	94,79059	49,37%
216000	Нет	26,15442	13,62%
216000	Да	94,83355	49,39%

Для программы, результаты работы которой указаны в таблицах 4 и 5, векторизация проводилась вручную, с использованием встроенных функций SSE. Использование векторизации дает ускорение в 3,6 раз, что близко к ожидаемому значению. Использование векторизации позволяет добиться эффективности решения задачи на ЦПУ примерно 50%.

Компилятор не сумел выполнить автоматическую векторизацию цикла, хотя внешний цикл не содержал зависимости по данным. Скорее всего, это объясняется сложностью используемого цикла: для достижения максимальной эффективности требуется выполнить векторизацию не только внутреннего цикла по j-частицам, но и внешнего.

Масштабируемость задачи на несколько ядер с использованием технологии OpenMP практически линейная, аналогично масштабируемости на большое число ядер в CELL.

На рисунке 4 приведена зависимость ускорения выполнения задачи на кластере от числа используемых ядер. За основу берется производительность задачи на одном узле, т.е. на 8 ядрах. Масштабируемость снова линейная, скорость выполнения ограничена производительностью ЦПУ, а не коммуникационной среды. Эффективность решения задачи на кластере примерно такая же, как и на одном узле, и составляет 50%.

Между отдельными запусками задачи, однако, может наблюдаться разброс по времени. Связано это с используемой схемой реализации, которая чувствительна к однородности скорости работы процессоров. Т.е. если в системе по каким-то причинам один из процессоров оказывается «медленным», то время решения задачи становится таким же, как если бы ее решал кластер, состоящий только из таких «медленных» процессоров. Причинами замедления могут быть неравномерность тактовой частоты, системный шум, другие задачи, работающие на том же процессоре и т.д. При этом по мере увеличения числа используемых узлов вероятность попадания на «медленный» процессор возрастает.

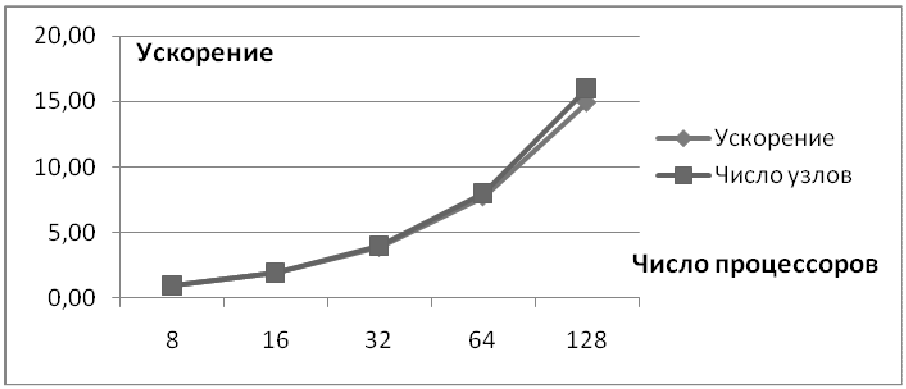

Рисунок 4. Масштабируемость задачи N тел на кластере.

5.5. Сравнение эффективности реализаций на различных архитектурах.

В таблице 6 сравнивается эффективность работы задачи N тел на различных архитектурах.

Таблица 6. Эффективность решения задачи N тел на различных архитектурах.

Архитектура	Достигнутая производительность, ГФлопс	Пиковая производительность, ГФлопс	Эффективность	Энергетическая эффективность, МФлопс/Вт
AMD HD4850	716,53	800	89,56%	1433
NVidia 8800GTX	211,60	340	62,24%	529
QS22 Blade Server	109,13	408	26,75%	503
«СКИФ-Чебышев», 1 узел	94,83	192	49,39%	114
«СКИФ-Чебышев», 16 узлов	1410,97	3072	45,93%	106

Можно видеть, что задача N тел с одинарной точностью наиболее эффективно решается с использованием графических процессоров. На данной задаче достигается от 60 до 90% пиковой производительности ГПУ, а реально достигаемая энергетическая эффективность достигает 1.4 ГФлопс/Вт. При этом энергетическая эффективность рассчитывается для всей системы, в которой большая часть энергии потребляется не ГПУ, а вспомогательным оборудованием – ЦПУ, материнской платой и т.д. По нашим оценкам, установка нескольких графических карт в одну машину позволит повысить энергетическую эффективность примерно в 2.5 раз.

Наиболее низкая загруженность достигается на процессоре CELL. Скорее всего, это связано с неэффективной работой компилятора spu-g++ даже при самых сильных настройках оптимизации. Достигаемая при этом производительность лишь ненамного превосходит ту, что достигается на «обычном» x64-процессоре. По нашим оценкам, однако, имеется возможность повысить эффективность примерно в 1.5 раза и достичь загруженности в 40%. По показателям энергетической эффективности CELL находится между ГПУ и обычными процессорами. При этом возможности сокращения потребления для CELL по сравнению с рассматриваемыми системами на основе ГПУ ограничены, поскольку QS22 Blade Server изначально создавался с целью минимизации энергопотребления, а поэтому большая часть возможностей уже использована.

Обычные процессоры, хотя и демонстрируют эффективность около 50%, сильно проигрывают нетрадиционным архитектурам по энергетической эффективности. По этому показателю они примерно в 4,5 раза хуже CELL и более чем на порядок уступают графическим процессорам.

При этом следует заметить, что традиционные компиляторы не могут эффективно транслировать гнездо из 2-х циклов, которое используется в задаче N тел, несмотря на отсутствие в нем зависимостей. Система C$ более эффективно справляется с этим гнездом циклов. Достигается это за счет использования альтернативного подхода: программа рассматривается не как

цикл, а как набор операций с массивами и функциями, при этом осуществляется поиск наиболее эффективного способа его отображения на целевую архитектуру. Массивы при этом доступны только на чтение или однократную запись. Кроме того, массивы рассматриваются как абстрактные типы данных, для которых способ представления на конкретной архитектуре может выбираться в зависимости от особенностей решаемой задачи. На других архитектурах более эффективные способы представления и отображения реализуются вручную.

Графические процессоры наиболее эффективны для решения задачи N тел только при использовании одинарной точности. Если для расчетов взаимодействия потребуется использовать двойную точность, показатели их энергетической эффективности снизятся примерно в 5 раз. В этом случае наиболее эффективной архитектурой для данной задачи будет процессор CELL; однако ГПУ все равно будут более эффективными, чем обычные процессоры.

На всех архитектурах производительность оптимизированной и неоптимизированной версий отличается не менее чем в несколько раз, а иногда и на порядки. При этом везде используются одинаковый набор оптимизаций:

- Векторизация (если есть векторные команды)
- Развертка цикла
- Использование иерархии памяти

Рассматриваемые в данной статье подходы к решению задачи N тел позволяют отметить ряд способов повышения эффективности программирования вычислительно емких задач.

- **Использование более сложных оптимизаций в компиляторе**. Используемые в C$ оптимизации могут быть перенесены и на другие вычислительные архитектуры. Их использование может существенно увеличить время компиляции; эффективным решением может быть использование директив компилятора, которые включали бы такую оптимизацию только для определенных участка кода.
- **Использование метапрограммирования в сочетании со специальными типами для работы с векторными типами и размещением данных в ОЗУ.** Данный подход требует больших усилий со стороны программиста. С другой стороны, он дает ему большие возможности оптимизации для конкретной архитектуры. Активное использование шаблонов и ООП позволит избежать дублирования кода и появления большого количества кода, работающего только на одной архитектуре. Однако сам код при этом становится менее читаемым.
- **Использование систем автоматической генерации и преобразования исходных кодов программ.**

6. Заключение.

В статье рассмотрены реализации вычислительного ядра задачи N тел на ряде современных вычислительных архитектур: процессоров x64, кластеров на основе процессоров x64, процессоров CELL, графических процессоров. Для каждой из архитектур рассматривается несколько возможных реализаций, начиная с простых и неэффективных, и заканчивая реализациями с высокой эффективностью. Для архитектур исследовано влияние различных оптимизаций на эффективность выполнения. При этом показано, что исследуемые архитектуры обладают множеством сходных признаков, и как следствие, для повышения эффективности программ на них используются сходные оптимизации. Также показано, что разница в производительности простых и оптимизированных программ может быть в несколько раз, а на некоторых архитектурах – на порядок. При этом оптимизацию часто требуется выполнять вручную, автоматическая векторизация или развертка циклов у современных компиляторов работает только в самых простых случаях.

На примере задачи N тел проведено сравнение различных архитектур. Показано, что для данной задачи, как и для большинства вычислительно емких задач, нетрадиционные архитектуры превосходят обычные процессоры по таким характеристикам, как производительность и энергетическая эффективность. При этом сложность достижения высокой производительности как на тех, так и на других может требовать длительной ручной оптимизации. В этих условиях

наличие общего языка программирования позволяет иметь единый код для нескольких архитектур, сокращая расходы на адаптацию и оптимизацию.

Литература

1. Elsen E. et al**.** N-Body Simulations on GPUs. // ACM/IEEE conference on Supercomputing 2006.
2. N-body simulations. *Scholarpedia.* [http://www.scholarpedia.org/article/N-body_simulations]
3. IBM Cell Broadband Engine Resource Center. [http://www.ibm.com/developerworks/power/cell/]
4. GPGPU.org. [http://www.gpgpu.org]
5. Makino J., Kokubo E., Fukushige T. Performance evaluation and tuning of GRAPE-6 - towards 40 "real" Tflops. // ACM/IEEE conference on Supercomputing-2003.
6. Makino J, Aarseth S. J. On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems // PASJ: Publications of the Astronomical Society of Japan, 1992, T. 44, pp. 141-151.
7. Barnes J., Hut P. A hierarchical O(N log N) force-calculation algorithm. 04.10.1986 r., Nature, T. 324, pp. 446 - 449.
8. Nitadori K., Makino J., Hut P**.** Performance Tuning of N-Body Codes on Modern Microprocessors: I. Direct Integration with a Hermite Scheme on x86_64 Architecture. [http://arxiv.org/abs/astro-ph/0511062]
9. Gualandris A., Zwart S. P., Tirado-Ramos A. Performance analysis of direct N-body algorithms for astrophysical simulations on distributed systems. [http://arxiv.org/abs/astro-ph/0412206]
10. Belleman R., Bedorf J., Zwart S. High Performance Direct Gravitational N-body Simulations on Graphics Processing Units II: An implementation in CUDA. [http://arxiv.org/abs/0707.0438v2]
11. Hamada T., Iitaka T. The Chamomile Scheme: An Optimized Algorithm for N-body simulations on Programmable Graphics Processing Units. [http://arxiv.org/abs/astro-ph/0703100]
12. Swaminarayan S. et al. 369 Tflop/s molecular dynamics simulations on the Roadrunner general-purpose heterogeneous supercomputer. // ACM/IEEE conference on Supercomputing-2008.
13. AMD FireStream 9270. [http://ati.amd.com/technology/streamcomputing/product_firestream_9270.html]
14. NVidia Tesla C1060. [http://www.nvidia.com/object/product_tesla_c1060_us.html]
15. Адинец А., Воеводин Вл.В. Графический вызов суперкомпьютерам // Открытые системы. 2008, №04, стр. 35-41.
16. Адинец, А. В., Кривов, М. А. Методы оптимизации программ для современных графических процессоров // Всероссийская научная конференция «Научный сервис в сети Интернет-2008: решение больших задач». стр. 345 – 353.
17. Процессоры Intel Xeon серий 5xxx. [http://parallel.ru/russia/MSU-Intel/xeons.html]
18. Intel Corp. Intel SSE4 Programming Reference. [http://softwarecommunity.intel.com/isn/Downloads/Intel%20SSE4%20Programming%20Reference.pdf]
19. NVidia CUDA Programming Guide v1.1. [http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf]
20. Warren M. et al. Pentium Pro Inside: I. A Treecode at 430 Gigaflops on ASCI Red, II. Price/Performance of $50/Mflop on Loki and Hyglac. // SuperComputing 1997.