COACCEL — конвейерная вычислительная среда для multi-GPU

Д.Н. Микушин[1], О.А. Левченко[2], А.П. Петров[3]

Институт Вычислительной Математики РАН[1], Геофизический Центр РАН[2],
Сибирский научно-исследовательский гидрометеорологический институт[3]

В данной работе представлена распределенная конвейерно-гибридная вычислительная среда COACCEL. Унификация программных интерфейсов взаимодействия составных частей гибридной системы в COACCEL позволяет масштабировать существующее GPU-приложение для массивно-параллельных систем. Управление потоками данных реализовано с помощью программного конвейера, стадии которого выполняют асинхронные операции на нескольких уровнях иерархии памяти. Анализ корректности выполнен на примере реализации разностных схем и функций BLAS. Результаты тестовых измерений приведены для распределенной среды, построенной из многоядерных процессоров и ускорителей GT200.

1. Введение

Доля гибридных суперкомпьютеров в последние годы существенно возросла. На конец 2010 г. в рейтинге top500 значится 28 систем гибридной архитектуры, когда как всего два года назад была лишь одна. В числе преимуществ подобных систем обычно называется большая вычислительная плотность в отношениях $/FLOPS и Ватт/FLOPS, когда как другая важная характеристика остаётся в тени: *программируемость*. Насколько трудоёмкой может оказаться разработка приложения с интенсивным обменом данными для такой системы, если потребовать чтобы эффективность реальных FLOPS приложения оставалась столь же привлекательной, как пиковые показатели и результаты на стандартных тестах?

Вокруг любой вычислительной технологии формируется экосистема средств разработки и вспомогательных библиотек. Наиболее развитая экосистема построена вокруг MPI и OpenMP, части которой перерабатываются для использования NVIDIA CUDA, OpenCL и Cell, формируя их собственные экосистемы. В результате гибридному суперкомпьютеру, основанному на комбинации каких-либо из перечисленных технологий, некоторые стандартные функции становятся доступными на всех уровнях. Однако сумма отдельных проработанных составляющих с точки зрения программируемости - это ещё не то же самое, что простая однородная система. Поэтому ответ на ранее поставленный вопрос зависит от того, насколько сочетаются друг с другом представления отдельных уровней о гибридной системе, и в какой мере их замкнутость возможно ослабить с помощью обобщений.

Рассмотрим гибридную систему, состоящую из многопроцессорных узлов с графическими ускорителями. Узлы соединены с помощью сети Infiniband, а массивы ускорителей подключены к параллельной PCI-Express шине узлов (рис. 1). Для облегчения восприятия карта Infiniband на схеме представлена ниже GPU, хотя она, вообще говоря, так же является PCI-E устройством[1]. Пунктирной линией на схеме показан путь, по которому выделенная, в качестве примера, пара GPU может обмениваться данными. Как видно, он состоит из звеньев (*интерфейсов*), соединяющих локальную память различных устройств.

$$GPU_1 \xrightarrow{cudaMemcpy} RAM_1 \xrightarrow{MPI\_Send/MPI\_Recv} RAM_2 \xrightarrow{cudaMemcpy} GPU_2 \qquad (1)$$

[1]Поскольку Infiniband - это PCI-E устройство, существует теоретическая возможность исключить из иерархии звенья, соответствующие памяти хостов и передавать данные напрямую по PCI-E шине, но такой режим в драйверах не релизован.

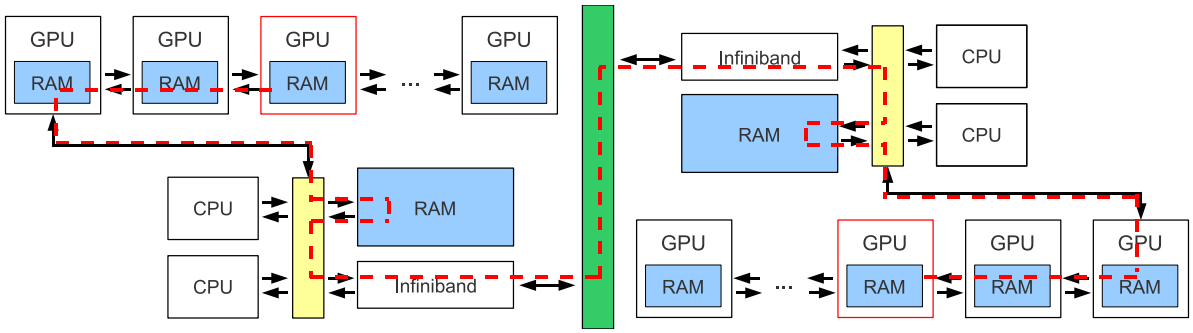

Рис. 1. Пара узлов гибридной вычислительной системы

Назовём совокупность интерфейсов и уровней памяти *иерархией памяти.*

Другой типичный пример — потоковая обработка данных, находящихся в памяти узла. Обычно для ускорения передачи данных из памяти узла по PCI-Express используется некешируемая (*nonpagable*) память, которая в случае большого объёма данных может присутствовать лишь в качестве дополнительного буфера. В этом случае полный цикл пересылки, обработки и возврата данных для каждого фрагмента будет состоять из четырёх стадий:

$$\begin{cases} RAM \xrightarrow{memcpy} RAM_{pinned} \xrightarrow{cudaMemcpyAsync} GPU, \\ GPU \xrightarrow{cudaMemcpyAsync} RAM_{pinned} \xrightarrow{memcpy} RAM \end{cases} \tag{2}$$

Из-за привязки контекстов устройств к потокам, каждая последовательность операций пересылки с участием GPU требует либо независимой обработки в отдельном потоке, либо переключения контекстов.

Ещё один аспект, усложняющий общую картину — асинхронность. Использование асинхронных обменов данными часто является единственным способом повышения эффективности вычислений, в случае если времена пересылок и расчётов соизмеримы. Это потребует в дополнение к многослойности функций обмена и потокам реализовывать в программе нити или конвейер.

Рассмотренные примеры показывают, что на каждом уровне памяти соответствующая технология задействует собственные механизмы обмена данными, которые, хоть и осложнены некоторыми особенностями, тем не менее функционально очень похожи. Из этих соображений получение общего единообразного интерфейса обмена данными действительно может позволить уменьшить сложность программирования гибридной системы.

Таким образом, целью настоящего исследования одновремено не является ни решение какой-либо узко специальной задачи, ни чрезмерно идеалистичное обобщение. Задача состоит в выработке базовых примитивов организации вычислений на гибридных системах, которые могут быть положены в основу промышленных библиотек и численных моделей, позволяя, к примеру, расширить действие библиотеки CUBLAS с одного GPU на несколько подобно тому, как в Intel MKL по сравнению с классическими BLAS/LAPACK реализована внутренняя многопоточность.

В следующих далее разделах статьи излагается функциональная суть некоторых базовых примитивов и алгоритмы их построения. Программная реализация представляет собой библиотеку COACCEL, для которой демонстрируются основные функции пользователя и пример производительности одного приложения. С целью получения большего числа экспертных оценок и привлечения сообщества к разработке тестовых приложений, приведены инструкции по загрузке актуальной версии исходного кода.

2. Постановка задачи

Логика выбора функциональных примитивов может быть основана на анализе некоторых свойств задач, эффективная параллельная реализация которых наиболее трудоёмка, таких как, например, случайный доступ к данным или разбаллансировка нагрузки, связанная с динамическим перестроением сеток. С другой стороны, нежелательно чрезмерно специализировать общий механизм в ущерб его применимости, например, передача библиотечной функции слишком большого контроля над управлением памятью или обменом данными потенциально ведёт к допущению лишь конкретного порядка следования элементов в памяти, т.е. привязке к типу сетки. Попытка полностью уйти от предметной специфики оставляет доступными для рассмотрения лишь несколько возможностей:

- Базовые свойства алгоритмов, например асинхронность
- Примитивы, на которых основана работа самой гибридной системы, например контекст или поток
- Синтетические конструкции — представление составных операций как новых базовых, например сведение (в представлении пользователя) рассмотренного выше обмена данными по иерархии памяти к простой команде асинхронного копирования

На этой основе сформулируем следующие задачи:

1. Предложить алгоритм и прототип системы одновременного исполнения GPU-программы на нескольких устройствах, управляемых в различных потоках хост-системы, каждый — эксклюзивно для конкретного GPU. Выделенной группе устройств создать хост-потоки для выполнения заданных пользователем функций инициализации, деинициализации и циклической обработки данных с полной или выборочной барьерной синхронизацией после каждой итерации.
2. Разработать алгоритмы и реализацию конвейера по иерархии памяти гибридной системы с поддержкой асинхронного копирования. Конвейер должен перемещать порции начального массива данных заданного размера по иерархии памяти, задаваемой последовательностью устройств. Если одно из устройств помечено как расчётное, то к данным, поступающим в его локальную память применяется заданная функция, и далее по иерархии отправляются изменённые данные. При копировании не должно делаться никаких допущений о структуре данных, вместо этого пользователю предлагается через обратный вызов самостоятельно формировать список сегментов данных, подлежащих пересылке, в зависимости от номера порции и уровня.
3. Используя предыдущие два примитива, построить общий явный метод разностного решения эволюционных уравнений на регулярной сетке с полуширинами разностного шаблона, задаваемыми с помощью параметров. Обобщение состоит вызове заданной функции пересчёта на каждом шаге по времени и автоматической синхронизации внутренних границ расчётной области (заданных ширин), образованных в результате декомпозиции.

Перечисленные задачи необходимо решить, предполагая, что в качестве ускорителей гибридной системы могут быть как устройства с поддержкой CUDA, так и OpenCL или других технологий.

3. Основные алгоритмы

3.1. Конвейерная пересылка данных

Аппаратные конвейеры присутствуют в процессорах и контроллерах, начиная едва ли не со времён самых ранних электронных систем 40-х годов. Принцип разделения составной операции на стадии асинхронного выполнения в длинном потоке команд позволяет минимизировать простой функциональных элементов. Если время *пролога* (начального насыщения функциональных элементов полезной работой) и *эпилога* много меньше времени конвейерной фазы (*pipeline steady state*), то можно считать, что общее время операции вместо бесконвейерного суммирования времён составляющих $\sum_i t_i$ стремится к $\max_i t_i$ — времени исполнения самой долгой операции.

Программные конвейеры (*software pipelining*) так же широко используются, в том числе для организации пересылки данных, например, в программных моделях CUDA [1] и Cell [2]. Однако чаще всего встречаются схемы, реализующие только самый простой вариант из трёх асинхронных операций — загрузка, расчёт и выгрузка. Для случая иерархической памяти данные требуется пропустить уже через несколько промежуточных уровней, общее число которых может изменяться.

Пусть данные длины L, поделенные на m порций равной длины l пересылаются с помощью конвейера по иерархии памяти из n уровней в прямом и обратном направлении. На каждом уровне памяти создадим пару из входящего и исходящего буфера длины l для обоих направлений движения данных. Тогда простой конвейер может быть реализован как цикл изменения состояний порций данных, проходящих стадии $LOAD$, $SAVE$, $SYNC$, $COMP$ на уровнях иерархии $\overline{0,n}$, в соответствии с правилами перехода (3).

3.2. Решение сеточных задач

Пусть в результате декомпозиции расчётой области на каждом из устройств получена часть результатов на очередном шаге по времени. Граничные полосы результирующих трёхмерных массивов подлежат синхронизации с результатами устройств, обрабатывающих соседние части расчётной области. Поскольку система гибридная, различные типы устройств могут использоваться для расчётов, например CPU и GPU. В таком случае для каждой пары устройств необходимо установить отдельный конвейер, а в случае двух GPU, работающих в разных потоках - два связанных конвейера для достижения ближайшей памяти хоста и обмена данными.

$$\begin{cases} & \to \quad LOAD^1, \\ LOAD^i & \to \quad SYNC^i, \\ SYNC^i & \to \quad LOAD^{i+1}, \qquad i = \overline{1, n-1}, \\ SYNC^n & \to \quad COMP^n, \\ COMP^n & \to \quad SAVE^n, \\ SAVE^i & \to \quad SYNC^i, \qquad i = \overline{n, 0} \\ SYNC^i & \to \quad SAVE^{i-1}, \qquad i = \overline{n, 1}. \end{cases} \tag{3}$$

- Если одно из устройств - это GPU, то конвейер должен работать в реверсивном режиме, т.е. стадии LOAD и SAVE меняются местами.

- Если устройства относятся к разным узлам, то на каждом узле может присутствовать

лишь одна часть конвейера, проходящая только по локальным узлам, но согласованная с удалённой идентификатором сообщений.

При динамическом построении конвейеров между заданными устройствами возникает задача поиска наилучшего пути по иерархии памяти, которую можно интерпретировать как поиск кратчайшего пути в заданном графе допустимых переходов между поддерживаемыми типамис устройств (может быть решена с помощью Алгоритма Дейкстры [3]).

4. Программная реализация

Для поддержки компонентов гибридной системы различного типа (в настоящий момент — CPU, CUDA и MPI в синхронных и асинхронных режимах) библиотека COACCEL реализует интерфейсы, представленные на Рис. 2. Чтобы добавить в библиотеку поддержку нового устройства, достаточно реализовать для него эти функции.

```
void* coaccel_memcpy_start(
        coaccel_device device, int count,
        coaccel_address* dst, coaccel_address* src,
        size_t* size, int dir);

void coaccel_memcpy_finish(coaccel_device device, void* desc);

void coaccel_device_lock(coaccel_device device);

void coaccel_device_unlock(coaccel_device device);

coaccel_address coaccel_device_malloc(
        coaccel_device desc, size_t size);
```

Рис. 2. Функции, реализуемые для устройств гибридной системы

Интерфейс *multi* (Рис. 3) реализует управление несколькими GPU в отдельных потоках, *pipeline* (Рис. 4) — конвейерную обработку с заданными пользователем функциями

```
typedef int (*coaccel_multi_func_t)(
        coaccel_device_group group, coaccel_device device,
        int idevice, int ithread, void* data);

coaccel_multi coaccel_multi_init(
        coaccel_device_group group, int nthreads,
        coaccel_multi_func_t thread_init,
        coaccel_multi_func_t thread_deinit,
        void* data);

void coaccel_multi_step_all(coaccel_multi desc,
        coaccel_multi_func_t process, void* data);

void coaccel_multi_finalize(coaccel_multi desc, void* data);
```

Рис. 3. Интерфейс *multi*, решение задачи №1

4.1. Тестовая задача

Примером работы реализованных интерфейсов является тест, основанный на функциях BLAS/CUBLAS, выполняющий для заданных матриц $L[m \times m]$, $K[m \times m]$, $R[m \times n]$, $Q[m \times n]$ последовательность матричных операций (4). Предполагается, что m достаточно мало, чтобы матрицы L и K помещались в локальную память каждого GPU, а n достаточно велико, чтобы матрицы R и Q можно было загрузить на GPU лишь частично. Расчёт выполняется по схеме поточной обработки (2), параллельно на заданном числе GPU.

```
typedef coaccel_memreq (*coaccel_pipeline_transfer_t)(
        int istage, size_t ihost, size_t idevice, size_t szchunk,
        void* data);

typedef int (*coaccel_pipeline_process_t)(
        int mode, size_t ihost, size_t idevice, void*);

coaccel_pipeline coaccel_pipeline_init(
        coaccel_device_group devices,
        size_t size, size_t szchunk, int reverse,
        coaccel_pipeline_transfer_t load_func,
        coaccel_pipeline_transfer_t save_func,
        coaccel_pipeline_process_t process_start_func,
        coaccel_pipeline_process_t process_sync_func, void* ptr);

void coaccel_pipeline_process(coaccel_pipeline desc);

void coaccel_pipeline_done(coaccel_pipeline desc);
```

Рис. 4. Интерфейс *pipeline*, решение задачи №2

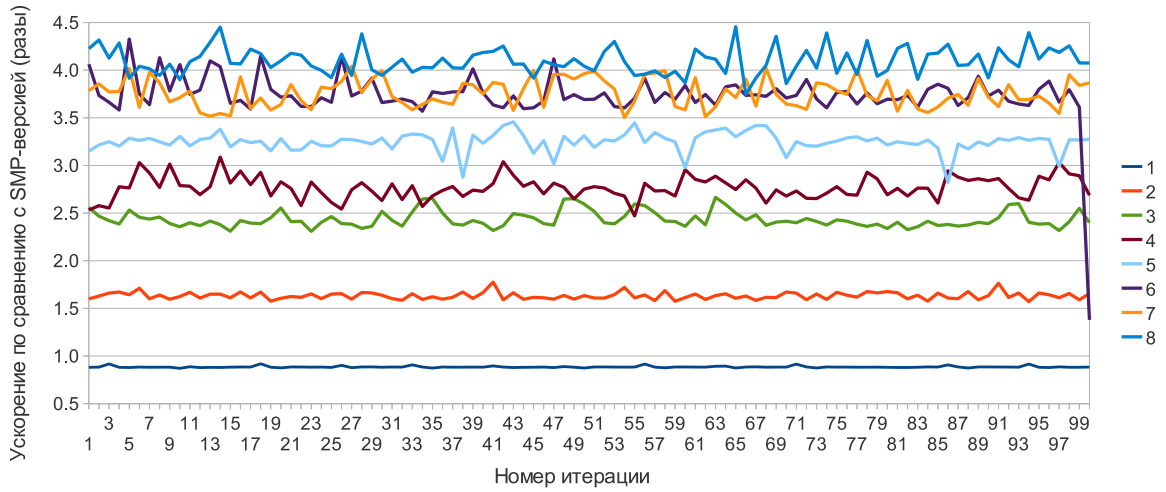

Рис. 5. Тестовая задача для гибридной SMP-системы Intel Xeon X5670 + 2×NVIDIA Tesla S1070 (8 GPUs)

$$\begin{aligned} R &:= 0.5(Q - R), & Q &:= 0.5(Q + R), \\ R &:= L \times R, & Q &:= K \times Q, \\ R &:= R + Q, & Q &:= R - Q. \end{aligned} \tag{4}$$

Тестирование проведено на сервере с двумя 6-ядерными процессорами Intel Xeon и двумя составными GPU S1070 (по 4 устройства архитектуры GT200 в каждом). На Рис. 5 представлена динамика изменения производительности теста в разах (ось Y) по сравнению с CPU-версией, использующей Intel MKL для каждой из последовательных 100 итераций метода (ось X). Каждая кривая f_{ngpus}, $ngpus = \overline{1,8}$ показывает наихудшую производительность GPU в соответствующей группе: $f_{ngpus}(t) = \min_i f_i(t)$, $i = \overline{1, ngpus}$ - номер GPU в группе, t - номер итерации. Проще говоря, время каждой итерации учитывает полную синхронизацию устройств.

Результаты данного теста получены для $m = 1024$, $n = 131072$, все вычисления проводились с двойной точностью. Примечательно, что обычная скорость пересылки данных до-

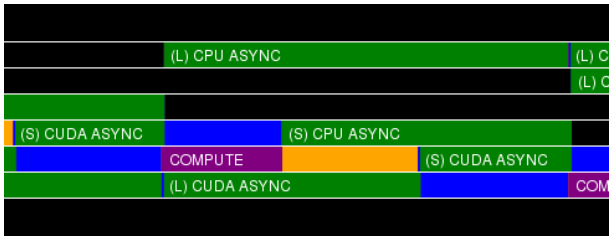

Рис. 6. Пример визуального представления работы программного конвейера (тестовая задача для гибридной SMP-системы).

стигается при задании в интерфейсе *pipeline* размера буфера $szchunk \geq 8$, что при четырёх таких буферах и дополнительных массивах в сумме составляет менее 64 Мбайт. Таким образом потоковая обработка оказывается совершенно не требовательной к размеру собственной памяти GPU. Однако необходимо учитывать, что в случае когда время пересылки данных становится сравнимым со временем вычислений, потери производительности с синхронным конвейером становятся значительными. На Рис. 6 визуализированы стадии обработки данных синхронного конвейера для того же самого теста, но на GPU Tesla C2050. Из графика видно, что если пересылки данных и вычисления не накладываются, то примерно половину времени устройство простаивает.

5. Заключение

В работе предложено несколько базовых конструкций, призванных сделать гибридный суперкомпьютер с графическими ускорителями («кластер из кластеров») более целостной системой с точки зрения программной модели обмена данными. По алгоритмам составлены программные ализации, корректность которых проверяется набором из 16 различных тестов, а быстродействие — тестовыми приложениями. Исходный код доступен по адресу [4].

Дальнейшее развитие проекта может быть связано с подготовкой подробной документации к примерам и разработкой большего числа демонстрационных приложений, в том числе для случаев асинхронного конвейера и сеточных задач, которые позволили бы лучше оценить применимость библиотеки в реальных условиях.

Литература

1. Shinta Nakagawa, Fumihiko Ino and Kenichi Hagihara. A middleware for efficient stream processing in CUDA // Computer Science - R&D, Vol. 25, Number 1-2, pp. 41-49, 2010.

2. Michael Kistler, Michael Perrone and Fabrizio Petrini. Cell Multiprocessor Communication Network: Built for Speed. // IEEE Micro, Vol. 26, pp. 10–23, 2006.

3. E. W. Dijkstra. A note on two problems in connection with graphs. // Numerische Mathematik. V. 1 (1959), Springer-Verlag, Berlin, P. 269-271

4. Микушин Д.Н. и др. COACCEL — the generalized framework for distributed computing on accelerators: URL: http://tesla.parallel.ru/trac/coaccel .