ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

С.А. Инютин, В.В. Чернобровкин

АДАПТАЦИЯ МОДУЛЯРНЫХ ВЫЧИСЛЕНИЙ ПОД GPU-ПРОЦЕССОРЫ С ПОМОЩЬЮ CUDA-ТЕХНОЛОГИИ

В работе рассматриваются приоритетные варианты распараллеливания многоразрядных вычислений, ориентированных на GPU-процессоры компании Nvidia, имеющие архитектуру, поддерживающую массивно-параллельные вычисления, и обладающие собственной памятью. Изложена методика адаптации модулярных вычислений под графические процессоры с помощью технологии CUDA. Введен алгоритм распараллеливания вычислений на уровне данных, основанный на непозиционной системе счисления. Совместное использование всех вышеперечисленных факторов дает следующее преимущество: значительное ускорение обработки большого объема вычисляемых данных.

GPU-процессоры, CUDA-технологии, система остаточных классов, китайская теорема об остатках, вычеты, распараллеливание процессов, производительность, многоразрядные данные.

В современной науке распараллеливание вычислений представляет собой широкое понятие проводимых одновременно и независимых друг от друга количественных вычислений. Причем само распараллеливание процессов вычислений можно проводить с учетом последних достижений в науке в виде следующих вариантов (рис. 1).

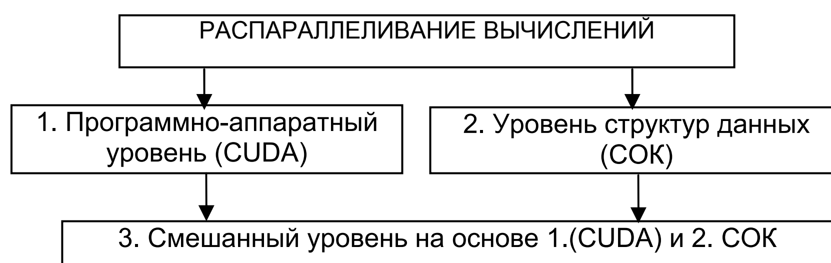

Рис. 1. Выбор распараллеливания вычислений

Программно-аппаратная архитектура массивно-параллельных вычислений CUDA позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia [3]. На рис. 1 СОК — это система остаточных классов (непозиционная система счисления)[1, 2, 4, 7, 8].

Рассмотрим GPU-процессоры на основе CUDA-технологии, которая, как известно, позволяет качественно распараллеливать многоразрядные вычисления (рис. 2).

Модель вычислительного устройства (ядра) GPU-процессора показывает, что верхний уровень ядра состоит из блоков одинакового размера, которые группируются в сетку (grid) размерностью $N_1 * N_2$.

Ядро GPU-процессора

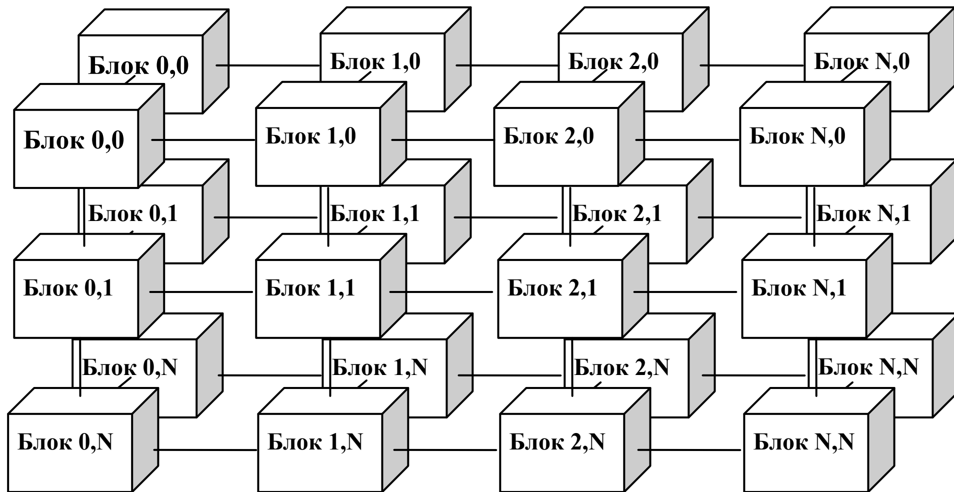

Рис. 2. Модель вычислительного устройства GPU-процессора

Каждый блок, в свою очередь, состоит из вычислительных нитей (threads), в которых непосредственно происходят вычисления. Нити в блоке могут быть сформированы в виде одномерного, двухмерного или трехмерного массива, показанного на рис. 3.

Блок ядра

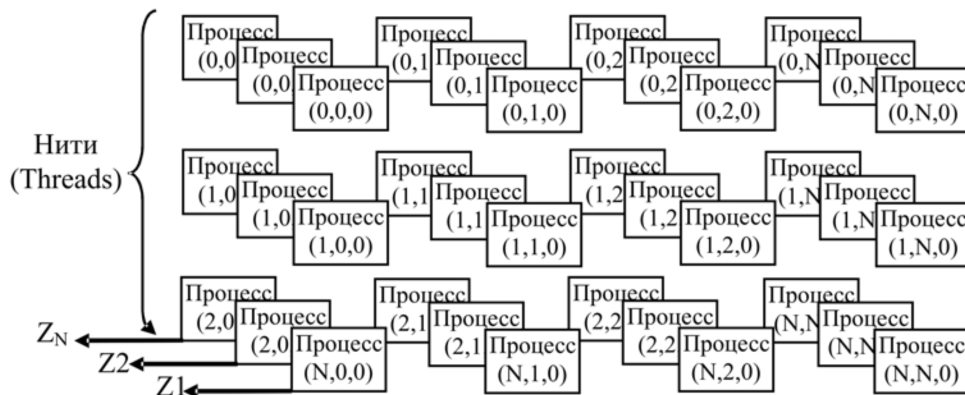

Рис. 3. Устройство блока ядра GPU-процессора

При использовании GPU-процессора можно задействовать грид необходимого размера и при помощи CUDA-технологии сконфигурировать блоки под параметры поставленной вычислительной задачи.

Каждое ядро GPU-процессора может работать одновременно с очень большим числом нитей, поэтому, чтобы ядро могло однозначно определить

номер нити, в CUDA используются встроенные переменные threadIdx и blockIdx.

При написании параллельного кода для GPU-процессора CUDA-технология имеет ряд дополнительных расширений языка С:

— Спецификаторы функций, которые показывают, как и откуда будут выполняться функции.

— Спецификаторы запуска ядра GPU.

— Спецификаторы переменных, которые служат для указания типа используемой памяти GPU.

— Встроенные переменные для идентификации нитей, блоков и др. параметров при исполнении кода в ядре GPU.

Спецификаторы функций определяют, как и откуда будут вызываться функции. Всего в CUDA три таких спецификатора:

— __host__ — выполняется на CPU, вызывается с CPU;

— __global__ — выполняется на GPU, вызывается с CPU;

— __device__ — выполняется на GPU, вызывается с GPU.

Спецификаторы запуска ядра служат, как упоминалось выше, для описания количества блоков, нитей и памяти, которые необходимо выделить при расчете на GPU-процессоре. Синтаксис запуска ядра имеет следующий вид:

myKernelFunc<<<gridSize, blockSize, sharedMemSize, cudaStream>>>(float*param1,float* param2),

где: *gridSize* — размерность сетки блоков (*dim3*), выделенной для расчетов,

blockSize — размер блока (dim3), выделенного для расчетов,

sharedMemSize — размер дополнительной памяти, выделяемой при запуске ядра,

cudaStream — переменная *cudaStream_t*, задающая поток, в котором будет произведен вызов.

Спецификаторы переменных *device*, *constant* и *shared* задают и размещают в памяти GPU-процессора переменные разных типов, которые, однако, имеют ряд ограничений.

Как упоминалось выше, общим приемом в CUDA-технологии является то, что исходная задача разбивается на набор отдельных подзадач, решаемых независимо друг от друга (рис. 4). Каждой такой подзадаче соответствует свой блок нитей.

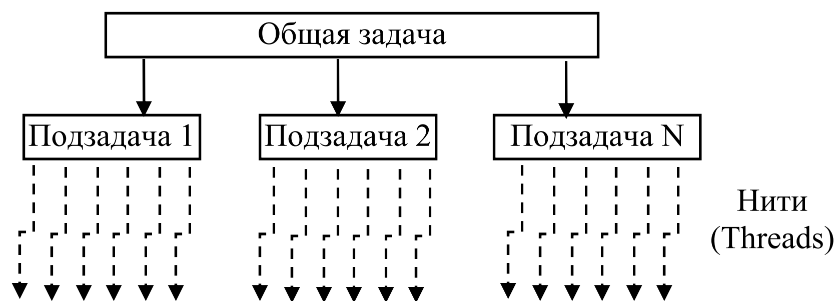

Рис. 4. Распараллеливание исходной задачи

Причем каждой отдельной нити соответствует один элемент вычислительных данных. Этот фактор говорит о том, что под такие условия можно легко адаптировать систему остаточных классов, обладающую естественным параллелизмом на уровне данных. Прежде чем к ней перейти, определим

производительность процессоров, которая измеряется в количестве выполняемых инструкций за определенное время.

$$P = \frac{N_{инс}}{T_{вып}}, \tag{1}$$

где P — производительность, $N_{инс}$ — количество инструкций, $T_{вып}$ — время выполнения вычислений.

Распишем процесс поподробнее, введем в формулу (1) тактовую частоту:

$$P = \frac{N_{\text{инс}}}{N_{\text{такт}}} \times \frac{N_{\text{такт}}}{T_{\text{вып}}}. \tag{2}$$

Как видно, первая часть полученного произведения — количество инструкций, выполняемых за один такт, вторая — количество тактов процессора в единицу времени (тактовая частота). Следовательно, для увеличения производительности нужно или поднимать тактовую частоту, или увеличивать количество инструкций, выполняемых за один такт. Но так как рост частоты может обернуться техническими проблемами, например перегрев процессора, то продуктивнее всего увеличивать количество исполняемых «за один такт» инструкций.

В свою очередь распараллеливание инструкций во многом зависит от среды программирования и необходимых алгоритмов.

Приведем простой пример: 1) A=[array1]; 2) B=[array2]; 3) C=A+B.

Первые две инструкции вполне можно выполнить параллельно, только третья от них зависит. А значит — всю программу можно выполнить за два шага, а не за три.

Максимальное ускорение, которое можно получить от распараллеливания программы на P процессоров (ядер), дается законом Амдала:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{P}}, \tag{3}$$

где P — число процессоров, α — последовательный фрагмент алгоритма.

Закон Амдала показывает, что прирост эффективности вычислений зависит от алгоритма задачи и ограничен сверху для любой задачи с $a \neq 0$. Однако не для всякой задачи имеет смысл наращивание числа процессоров в вычислительной архитектуре. Более того, если учесть время, необходимое для передачи данных между узлами вычислительной системы, то зависимость времени вычислений от числа узлов будет иметь максимум. Это накладывает ограничение на масштабируемость вычислительной системы, то есть означает, что с определенного момента добавление новых узлов в систему будет увеличивать время расчета задачи.

Оптимизация производительности, как правило, сводится к следующим шагам:

1) максимальное использование параллелизма задачи;
2) оптимизация доступа в память;
3) оптимизация машинной арифметики.

На первом шаге необходимо решить проблему «последовательного участка» (п. 3) параллельного алгоритма. Возможные варианты решения — это уменьшение такого участка либо попытка преобразовать его большую часть в параллельный код. Второй шаг подразумевает разработку наименее затрат-

ного варианта обращения к памяти. На третьем шаге необходимо разработать и применять вычислительные алгоритмы, построенные на модулярных структурах данных [7].

Перейдем непосредственно к параллельным алгоритмам, которые основаны на модулярной системе счисления или, как упоминалось выше, системе остаточных классов [5]. Основной теоретико-числовой базой системы остаточных классов является теория сравнений [8]. Система остаточных классов дает нестандартное представление чисел, иначе говоря, в данной системе числа представляются своими остатками от деления на выбранную систему оснований и все рациональные операции могут выполняться параллельно над цифрами каждого разряда в отдельности.

Для понимания вычислительных операций в модулярной системе докажем один из вариантов Китайской теоремы об остатках (КТО) [2, 4, 5, 6, 7], которая является фундаментальным положением в основе модулярного представления чисел.

Теорема 1 (Китайская теорема об остатках). Любое целое положительное число *А* в модулярной системе счисления можно представить в виде набора остатков (вычетов) от деления этого числа на выбранные основания (модули) системы.

Доказательство. Пусть число *А* представлено в системе остаточных классов как:

$$A \leftrightarrow (a_1, a_2, ...a_n), \tag{4}$$

где $a_1, a_2, ...a_n$ — наименьшие, неотрицательные вычеты, образованные путем целочисленного деления числа *А* на выбранные основания

$$P_i = p_1 * p_2 * ... * p_n = \prod_{i=1}^{n} p_i , \tag{5}$$

где p_i — взаимно простые числа

И, если $(p_i, p_j) = 1,$ т.е. взаимно простые, то представление числа (4) является единственным, при условии

$$0 \le A \le P_i. \tag{6}$$

Тогда число *А* будет выглядеть следующим образом:

$$\begin{aligned} &A \equiv a_1(\text{mod } p_1);\\ &A \equiv a_2(\text{mod } p_2)\\ &......................\\ &A \equiv a_n(\text{mod } p_n) \end{aligned} \tag{7}$$

Теорема доказана.

Формула (7) означает сравнения, которые показывают, что каждый вычет a_i получается независимо, а значит параллельно от других, и содержит информацию обо всём числе. Данное обстоятельство определяет: если для таких вычислений организовать программно-аппаратную вычислительную базу, то вычеты могут образовываться параллельно.

Цифры a_i представления (4) по выбранным модулям образуются следующим образом:

$$a_i = A(\mathrm{mod}\, p_i) = A - \left|\frac{A}{p_i} p_i\right|, \quad (\forall i \in [1,...n]), \tag{8}$$

где $\left|\frac{A}{p_i}\right|$ — целочисленное частное; p_i — взаимно простые числа (основания системы).

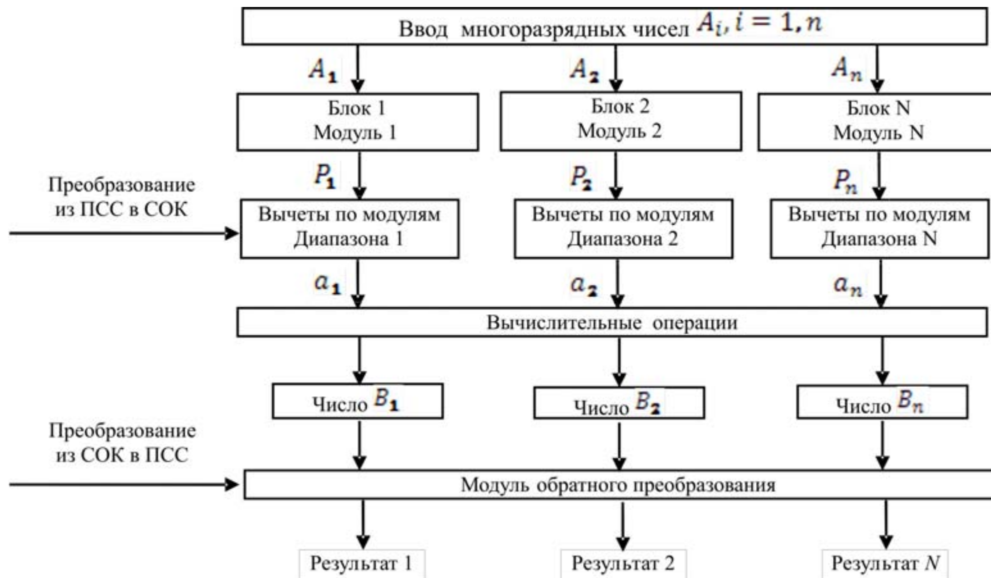

Рис. 5. Абстрактная вычислительная схема адаптации модулярных вычислений на GPU-процессор

Выводы

1) Архитектура CUDA приводит к двум парадигмам параллельного программирования на уровне задач и уровне данных.

2) В рамках подхода, основанного на параллелизме задач, для каждой подзадачи пишется своя собственная программа, где все они должны обмениваться результатами своей работы, получать исходные данные. Программист может контролировать и распределять подзадачи между различными процессорами.

3) Параллелизм на уровне данных может представлять собой модулярные алгоритмы, хорошо подходящие для многоразрядных вычислений.

4) Ускорение, заданное законом Амдала, можно повысить за счет эффективно написанных инструкций, основанных на идее модулярного представления вычислений.

ЛИТЕРАТУРА

1. *Амербаев В.М.* Теоретические основы машинной арифметики. Алма-Ата: Наука, 1976. 320 с.

2. *Боресков А.В., Харламов А.А.* Основы работы с технологией CUDA. М.: ДМК Пресс, 2010. 17 с.

3. *Бухштаб А.А.* Теория чисел. М: Наука, 1975.

4. *Василенко О.Н.* Теоретико-числовые алгоритмы в криптографии. МЦНМО, 2003. 294 с.

5. *Инютин С.А.* Вычислительные задачи большой алгоритмической сложности и модулярная арифметика // Вестник Тюменского государственного университета. № 3. 2002. С. 3–10.

6. *Червяков Н.И., Сахнюк П.А., Шапошников А.В., Ряднов С.А.* Модулярные параллельные вычислительные структуры нейропроцессорных систем. М.: Физматлит, 2003. 43 с.

7. *Чернобровкин В.В.* Распараллеливание вычислительных операций на основе модулярных структур данных // Современные проблемы науки и образования [Электрон. журнал]. № 11. 2013. С. 3–7.

8. *Чебышев П.Л.* Теория сравнений. СПб.: ИАН, 1849.

S.A. Inutin, V.V. Chernobrovkin

ADAPTATION OF MODULAR COMPUTING FOR GPU PROCESSORS WITH THE HELP OF CUDA TECHNOLOGY

There are considered the priority options parallelization multi-bit computing-oriented GPU processors Nvidia with the architecture ru supported massively parallel computing and having its own memory. The technique of adaptation of modular computing under GPUs with CUDA technology. Introduced algorithm parallelization of calculations on the data tier is based on non-positional number system. Sharing all the above factors gives the following advantage: significant acceleration the processing of large amount of calculated data.

GPU processors, CUDA technology, the system of residual classes, Chinese theorem on residues, deductions, paralleling processes, performance, multibit data.