МГУ им. М.В.Ломоносова

Шувалов Антон Юрьевич

5 курс, группа 525, кафедра вычислительной механики

Реализация алгоритма численного решения уравнений Навье-Стокса с использованием графического процессора.

Implementation of the algorithm for numerical solutions of the Navier-Stokes equations using a graphics processor

Курсовая работа

Научный руководитель:

доктор физ.-мат. наук,

профессор Луцкий А.Е.

Механико-математический факультет МГУ, 2020

Оглавление

Оглавление

Введение

Расчетная сетка - совокупность точек (узлов сетки), заданных в области определения некоторой функции. Расчетные сетки используются при численном решении дифференциальных и интегральных уравнений. Качество построения расчетной сетки в значительной степени определяет успех численного решения уравнения. Точность решения может быть повышена путем уменьшения размеров элементов сетки. Однако, уменьшение размеров ячеек во всей области значительно увеличивает вычислительную сложность задачи. Большинство задач математической физики требуют огромной вычислительной мощности. Отчасти эта проблема решается, если перейти к параллельным вычислениям. Параллельная программа – это ансамбль взаимодействующих слабосвязанных последовательных процессов. Создание комплексов программ, которые могут выполнять расчеты на параллельных компьютерах является достаточно сложной и трудоемкой задачей. Однако, есть и еще один способ решения данной проблемы, который состоит в том, чтобы переложить часть вычислений на графический процессор, который способен выполнять большие вычисления за меньшие промежутки времени. Именно в этом состояла моя задача в данной работе с конкретной программой.

Описание последовательного кода

Уравнения Навье-Стокса и Рейнольдса с двумя пространственными переменными при использовании гипотезы Буссинеска могут быть записаны в единообразной форме.

$$\frac{\partial U}{\partial t}+\frac{\partial F}{\partial x}+\frac{\partial G}{\partial y}=H$$

$$U=(\rho,\rho u,\rho v,e)^T$$

$$F=F^i+F^v, G=G^i+G^v, H=\frac{\omega}{y}(H^i+H^v)$$

$$F^i=(\rho u,\rho u^2+p,\rho u\upsilon,(e+p)u)^T,$$

$$F^v=(0,-\tau_{xx},-\tau_{xy},-u\tau_{xx}-\upsilon\tau_{xy}-q_x)^T$$

$$G^i=(\rho\upsilon,\rho u\upsilon,\rho\upsilon^2+p,(e+p)\upsilon)^T,$$

$$G^v=(0,-\tau_{xy},-\tau_{yy},-u\tau_{xy}-\upsilon\tau_{yy}-q_y)^T$$

$$H^i=(-\rho\upsilon,-\rho u\upsilon,-\rho\upsilon^2,-(e+p)\upsilon)^T,$$

$$H^v=(0,\tau_{xy},\tau_{yy}-\tau_{\varphi\varphi},u\tau_{xy}+\upsilon\tau_{yy}+q_y)^T$$

В нашем случае решается плоская задача обтекания цилиндра, симметричная относительно оси, проходящей через центр цилиндра и направленной вдоль направления потока, поэтому $\omega=1$.

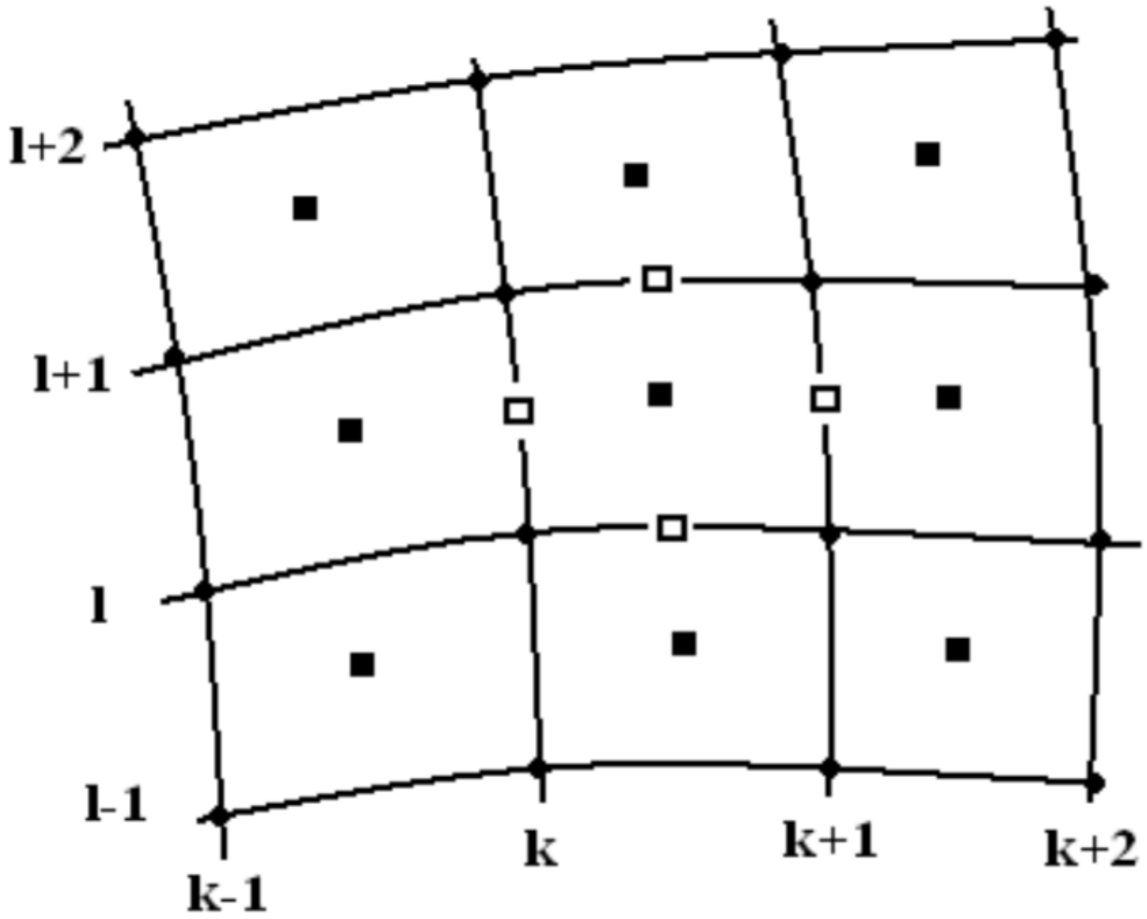


Схема расчета ячейки

Рассмотрим аппроксимацию потоков на примере ребра с номером k+1,l+1/2 (см. рис) . Пусть $U_{\mathrm{k+1/2,l+1/2}}$ - величины, отнесенные к центрам ячеек, $U_{k,l} = (U_{k-1/2,l-1/2} + U_{k+1/2,l-1/2} + U_{k+1/2,l+1/2} + U_{k-1/2,l+1/2})/4$ - величины в узлах сетки.

Производные $(U_x, U_y)_{\mathrm{k+1,l+1/2}}$, входящие в выражения потоков, вычисляются через разности $(U_{k+3/2,l+1/2} - U_{k+1/2,l+1/2})$ и $(U_{k+1,l+1} - U_{k+1,l})$.

Значения функций $U_{\mathrm{k+1,l+1/2}}$ на ребре определяются из решения задачи Римана с начальными данными

$$U^{+}_{k+1} = U_{k+1/2,l+1/2} + 1/2\Delta x U_{x,k+1/2,l+1/2} + 1/2\Delta y U_{y,k+1/2,l+1/2}$$

$$U_{k+1}^{-} = U_{k+3/2,l+1/2} - 1/2\Delta x U_{x,k+3/2,l+1/2} - 1/2\Delta y U_{y,k+3/2,l+1/2}$$

Для обеспечения монотонности разностной схемы производные в ячейках определяются в соответствии с принципом минимума модуля производных на противоположных ребрах

$$U_{x,k+1/2,l+1/2} = \min mod(U_{x,k,l+1/2}, U_{x,k+1,l+1/2})$$

$$U_{y,k+1/2,l+1/2} = \min mod(U_{y,k,l+1/2}, U_{x,k+1,l+1/2})$$

Таким образом, аппроксимация конвективных членом аналогична разностной схеме В.П.Колгана [3].

Структура последовательного кода:

1. Input:

1.1. Размер сетки

1.2. Список всех координат

1.3. Начальные значения газодинамических величин в углах сетки

1.4. Остальные параметры, характеризующие данную задачу (максимальное число шагов, шаг по времени, начальное время)

2. Step (в цикле по времени) :

2.1. Проводятся расчеты газодинамических величин с учетом геометрии изучаемого объекта для следующего момента времени

2.2. При необходимости производится вывод промежуточных параметров

2.3. Осуществляется адаптация расчетной сетки

3. Output:

3.1. Результаты вычислений газодинамических величин

Реализация алгоритма с использованием архитектуры CUDA

CUDA— программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

Архитектура CUDA даёт возможность по своему усмотрению организовывать доступ к набору инструкций графического или тензорного ускорителя и управлять его памятью. Функции, ускоренные при помощи CUDA, можно вызывать из различных языков, в т.ч. Python, MATLAB и т.п.

Первоначальная версия CUDA SDK была представлена 15 февраля 2007 года. В основе интерфейса программирования приложений CUDA лежит язык Си с некоторыми расширениями. Для успешной трансляции кода на этом языке в состав CUDA SDK входит собственный Си-компилятор командной строки nvcc компании Nvidia. Компилятор nvcc создан на основе открытого компилятора Open64 и предназначен для трансляции host-кода (главного, управляющего кода) и device-кода (аппаратного кода) в объектные файлы, пригодные в процессе сборки конечной программы или библиотеки в любой среде программирования. Учёные и исследователи широко используют CUDA в различных областях, включая астрофизику, вычислительную биологию и химию, моделирование динамики жидкостей, электромагнитных взаимодействий, компьютерную томографию, сейсмический анализ и многое другое. Однако у архитектуры CUDA есть свои недостатки:

- отсутствие поддержки рекурсии для выполняемых функций;
- минимальная ширина блока в 32 потока;

Терминология CUDA

NVIDIA оперирует весьма своеобразными определениями для CUDA API. Они

отличаются от определений, применяемых для работы с центральным процессором.

•Хост(Host) - центральный процессор, управляющий выполнением программы.

• Устройство(Device)—видеоадаптер, выступающий в роли сопроцессора центрального процессора.

• Грид(Grid)—объединение блоков, которые выполняются на одном устройстве.

• Блок(Block)—объединение тредов, которое выполняется целиком на одном SM. Имеет свой уникальный идентификатор внутри грида.

• Тред(Thread,поток)—единица выполнения программы. Имеет свой уникальный идентификатор внутри блока.

• Варп(Warp)—32 последовательно идущих треда, выполняется физически одновременно.

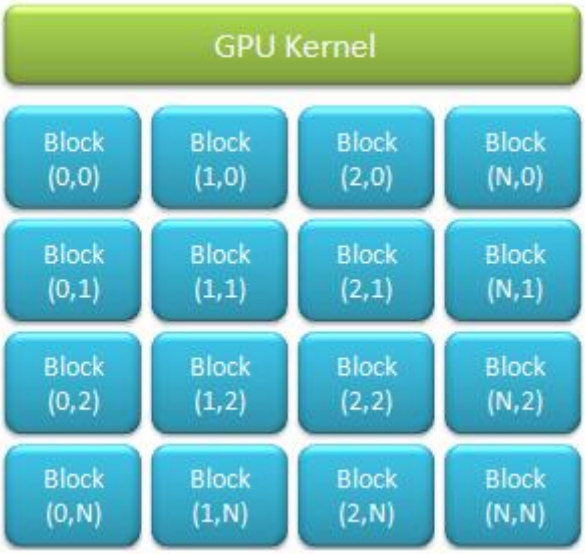
• Ядро(Kernel)—параллельная часть алгоритма, выполняется на гриде.

Такое разделение данных применяется исключительно для повышения производительности. Так, если число мультипроцессоров велико, то блоки будут выполняться параллельно. Если же карта не рассчитана на сложные вычисления (разработчики рекомендуют для сложных расчетов использовать адаптер не ниже уровня GeForce 8800 GTS 320 Мб), то блоки данных обработаются последовательно.

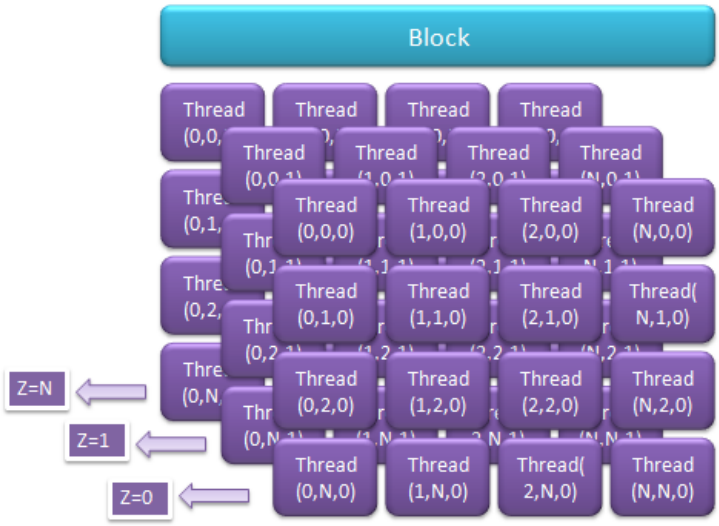
Вычислительная модель GPU

Верхний уровень ядра GPU состоит из блоков, которые группируются в сетку или грид (grid) размерностью N1 * N2 * N3. Размерность сетки блоков можно

узнать с помощь функции cudaGetDeviceProperties, в полученной структуре за это отвечает поле maxGridSize.



Любой блок в свою очередь состоит из нитей (threads), которые являются непосредственными исполнителями вычислений. Нити в блоке сформированы в виде трехмерного массива (рис. 2), размерность которого так же можно узнать с помощью функции cudaGetDeviceProperties, за это отвечает поле maxThreadsDim.



CUDA и язык C

Сама технология CUDA (компилятор nvcc.exe) вводит ряд дополнительных расширений для языка C, которые необходимы для написания кода для GPU:

1. Спецификаторы функций, которые показывают, как и откуда будут выполняться функции.
2. Спецификаторы переменных, которые служат для указания типа используемой памяти GPU.
3. Спецификаторы запуска ядра GPU.
4. Встроенные переменные для идентификации нитей, блоков и др. параметров при исполнении кода в ядре GPU .
5. Дополнительные типы переменных.

Теперь, после небольшого введения в особенности архитектуры CUDA, перейдем к рассмотрению конкретной задачи, поставленной передо мной. Из-за отличной от MPI системы работы в CUDA нет необходимости разбиения расчетной области на непересекающиеся подобласти. Однако в процессе разработки возникают другие трудности. Появляется необходимость передачи данных между CPU и GPU. После чего проводятся расчеты данной задачи и передача данных обратно для вывода. Общая схема действий, необходимых для выполнения расчетов на графической карте, выглядит следующим образом:

1. Получить данные для расчетов.
2. Скопировать эти данные в GPU память.
3. Произвести вычисление в GPU через функцию ядра.
4. Скопировать вычисленные данные из GPU памяти в CPU.
5. Посмотреть результаты.
6. Высвободить используемые ресурсы.

Для выделения памяти на видеокарте используется функция cudaMalloc, которая имеет следующий прототип:

cudaError_t cudaMalloc(void** devPtr, size_t count), где

- devPtr – указатель, в который записывается адрес выделенной памяти,
- count – размер выделяемой памяти в байтах.

Для копирования данных в память видеокарты и обратно используется cudaMemcpy, которая имеет следующий прототип:
cudaError_t cudaMemcpy(void* dst, const void* src ,size_t count, enum cudaMemcpyKind kind), где

- dst – указатель, содержащий адрес места-назначения копирования,
- src – указатель, содержащий адрес источника копирования,
- count – размер копируемого ресурса в байтах,
- cudaMemcpyKind – перечисление, указывающее направление копирования (может быть cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice).

Приведу пример кода, вписанного мной, для работы одной функции на графической карте (так как CUDA лучше работает с одномерными массивами, пришлось произвести работу, по приведению всех массивов к векторам).

```
double *gdf_CUDA, *dxf_CUDA, *dyf_CUDA, *ndu_CUDA, *nlr_CUDA, *w_CUDA, *tlr_CUDA, *fllr_CUDA, *fldu_CUDA, *tdu_CUDA, *gdf__CUDA;

tlr_O = (double*)malloc(len * 2 * sizeof(double));

Copy2DTO1D(tlr, tlr_O, 2, len);

cudaMemcpy(tlr_CUDA, tlr_O, len * 2 * sizeof(double), cudaMemcpyHostToDevice);

__global__ void derivs(double *dxf_O, double *dyf_O, double *gdf_O, double *w_O, double *nlr_O, double *ndu_O, double *vol, double mu_ref, double *tauT, int IT, int JT, int i_qgd, int Blocks, int Threads)

//Объявление необходимых переменных
```

```
for (i = blockIdx.x + 2; i < IT - 2; i += Blocks)

            for (j = threadIdx.x + 2; j < JT - 2; j += Threads){

//Проведение расчетов

}
```

cudaMemcpy(gdf_O, gdf_CUDA, len * 6 * sizeof(double), cudaMemcpyDeviceToHost);

Проведем разбор написанного выше.

```
double *gdf_CUDA, *dxf_CUDA, *dyf_CUDA, *ndu_CUDA, *nlr_CUDA, *w_CUDA, *tlr_CUDA, *fllr_CUDA, *fldu_CUDA, *tdu_CUDA, *gdf__CUDA;

cudaMalloc((void**)&gdf_CUDA, len * 6 * sizeof(double));
```

Объявление переменных и выделение памяти на видеокарте для расчета с использованием CUDA.

```
tlr_O = (double*)malloc(len * 2 * sizeof(double));

Copy2DTO1D(tlr, tlr_O, 2, len);

cudaMemcpy(tlr_CUDA, tlr_O, len * 2 * sizeof(double), cudaMemcpyHostToDevice);
```

Выделение памяти для перевода двумерных массивов в одномерные.

Непосредственный перевод с использованием написанной мной функции.

Копирование данных на видеокарту.

```
__global__ void derivs(double *dxf_O, double *dyf_O, double *gdf_O, double *w_O, double *nlr_O, double *ndu_O, double *vol, double mu_ref, double *tauT, int IT, int JT, int i_qgd, int Blocks, int Threads)
```

Объявление функции с атрибутом __global__ , который указывает, что запуск этой функции будет проводиться на ядре.

```
for (i = blockIdx.x + 2; i < IT - 2; i += Blocks)

            for (j = threadIdx.x + 2; j < JT - 2; j += Threads){
```

Разбиение расчетных циклов таким образом, чтобы каждая нить выполняла примерно одинаковое количество вычислений.

```
cudaMemcpy(gdf_O, gdf_CUDA, len * 6 * sizeof(double), cudaMemcpyDeviceToHost);
```

Копирование данных на CPU для последующего вывода.

Первичные попытки проверки работоспособности давали ускорение примерно в 4.7 раз, но, прочитав дополнительную документацию, выяснилось, что ускорение растет с увеличением количества узлов сетки и что наилучший результат ускорения получается при количестве потоков равном 128 или 256 штук (рекомендуется разработчиками компании Nvidea). Было решено увеличить количество ячеек сетки в 4 раза и использовать рекомендуемое количество потоков. Количество блоков же взято так, чтобы было возможно такое распределение расчетов между потоками, чтобы каждый из них выполнял вычисления для одной ячейки расчетной сетки, если ее масштабы позволяли сделать это (максимальное количество блоков равно 65,535). Иначе увеличивалось количество ячеек сетки на один поток, пока условие на максимальное количество блоков не было выполнено. Проделанные шаги позволили получить ускорение примерно в 6 раз.

Кроме того результативность метода ускорения с использованием технологии CUDA сильно зависит от масштаба расчетной сетки. Так как внедрение GPU подразумевает распределение задачи между центральным и графическим процессором. Производительность работы с центральным процессором в большей степени характеризуется его мощностью. А производительность видеокарты характеризуется не только мощностью графического процессора, но и скоростью передачи данных из оперативной памяти в видеопамять. Это связано с тем, что перед проведением вычислений на видеокарте требуется передать на нее данные, а затем получить их обратно. Однако, из-за того, что данные передаются на

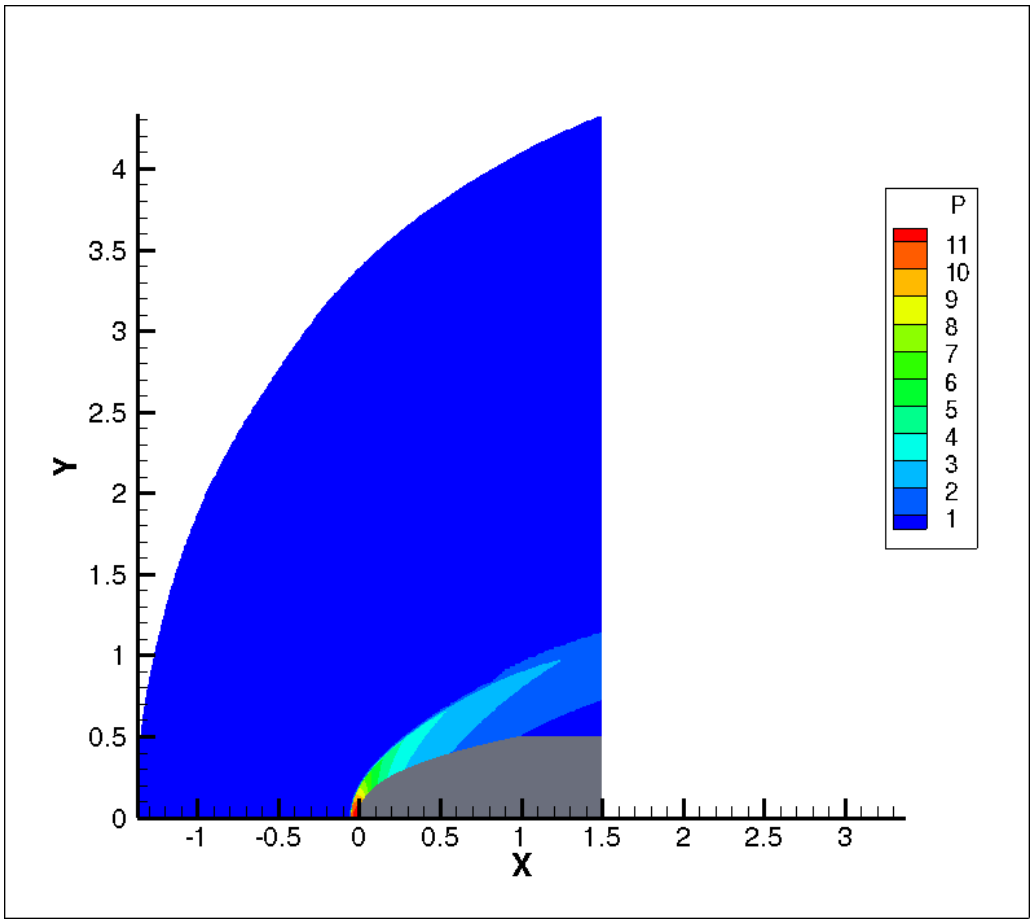
видеокарту с малой скоростью, время работы с этим устройством значительно увеличивается. Поэтому процесс передачи данных, по сути, является потерей времени.

Очевидно, что чем меньше операций будет приходиться на единицу передаваемых данных, тем менее выгодным будет использование графического процессора.

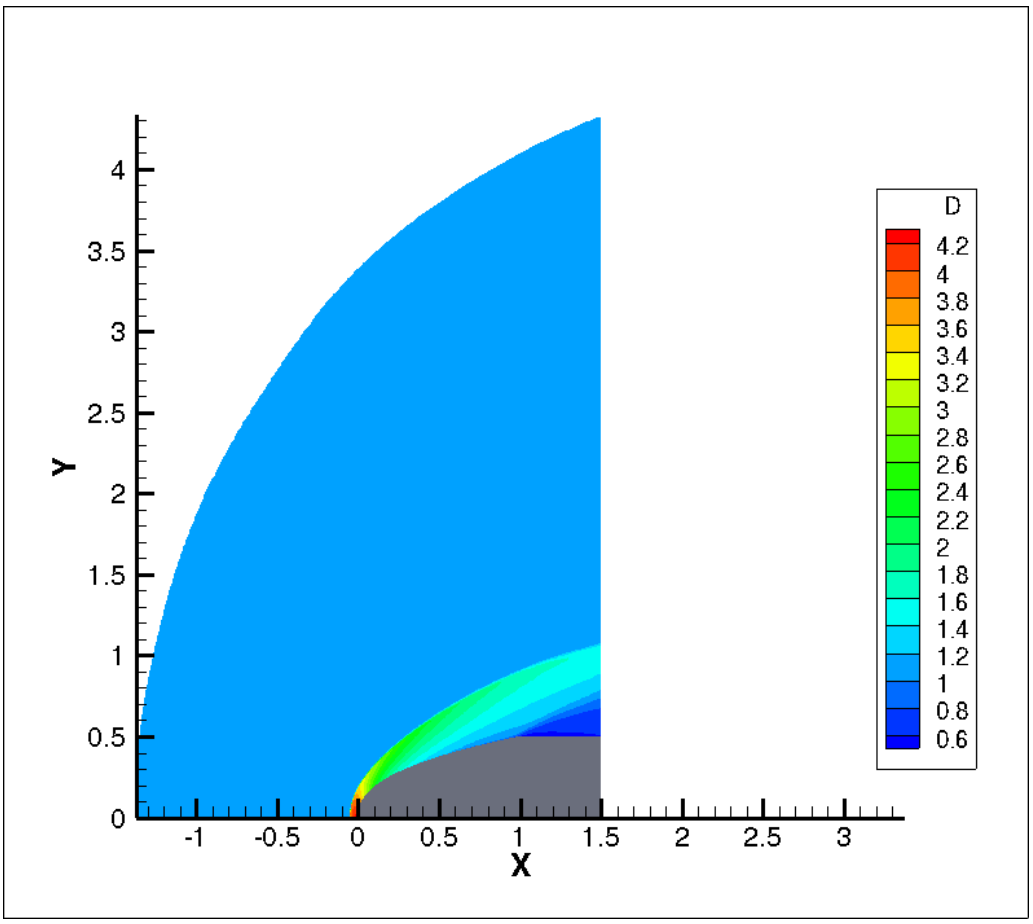
Примеры расчетов

После написания кода были проведены несколько расчетов с использованием CPU и GPU с целью сравнения результатов работы и времени выполнения (для уменьшения затрат времени количество шагов по времени было ограничено 5000).

Далее предоставлен результат расчета для симметричной задачи обтекания цилиндра потоком. В связи с симметричностью задачи на графиках продемонстрирована только область, находящаяся выше плоскости симметрии.



Обтекание цилиндра, поля давления.



Обтекание цилиндра, поля плотности.

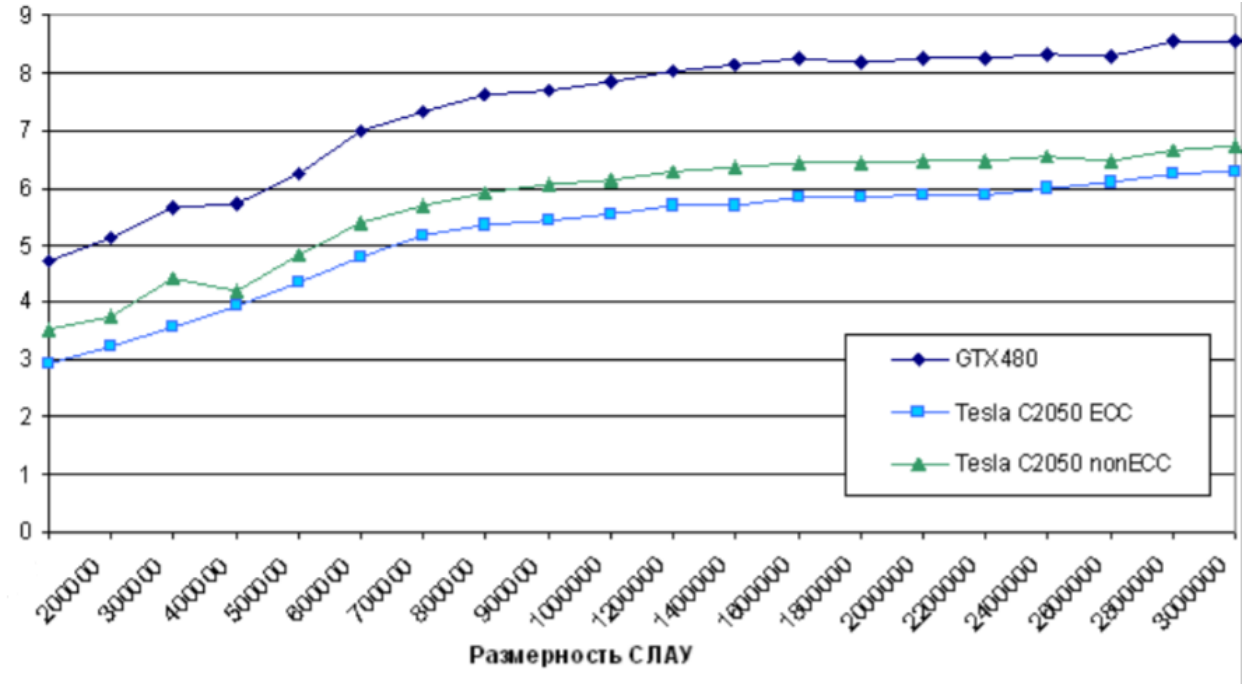
Получены следующие результаты замеров времени при запуске на:

Оригинальном коде	Коде с использованием CUDA
0,59	0,1

Возможно, полученное ускорение примерно в шесть раз, может показаться недостаточно хорошим результатом, но в большинстве статей, прочитанных мною по этому поводу, среднее ускорение как раз и было в 5-6 раз (примеры статей приведены в списке источников под пунктами 9 и 10).

В работе [9] было описано использование видеокарт для выполнения вычислений при решении задач строительной механики методом конечных элементов. Авторы находили внутренние усилия и деформации конструкций от заданной внешней нагрузки. Был использован метод конечных элементов,

В качестве алгоритма решения СЛАУ использовался метод сопряженных градиентов с предобуславливателем Якоби.
Результаты ускорения кода на графической плате относительно кода на CPU разнятся в зависимости от использованной видеокарты и количества ячеек расчетной сетки. Подробные результаты представлены на графике ниже.



Во второй же работе (номер 10 в списке источников) авторы рассматривали время, необходимое CPU и GPU для следующих методов разложения матрицы: LU-разложение, метод Холецкого и QR-разложение. Измерения производились на центральном процессоре Intel Core2 Quad Q6850 (4 ядра, 3.0 ГГц) и двух видеокартах: NVIDIA GeForce 8800GTX (128 ядер, 575 МГц, 768 Мб видеопамяти) и NVIDIA GeForce GTX280 (240 ядер, 602 МГц, 1Гб видеопамяти). Результаты представлены в таблице ниже.

Видеокарта	8800GTX	GTX280
Алгоритм	Ускорение	Ускорение
LU-разложение	2,5	4,1
Холецкий	2,7	4,4
QR-разложение	2,6	4,4

Видно, при использовании видеокарты NVIDIA GeForce 8800GTX удалось ускорить код более чем в 2.5 раза, а для NVIDIA GeForce GTX280 – более чем в 4.1 раза.

Кроме того из этих статей мы видим, что ускорение сильно зависит от используемого оборудования. В аппарате, на котором проводились вычисления данной курсовой работы, стоит только встроенная видеокарта (без дискретной), что также могло отрицательно сказаться на результате ускорения.

Заключение

1. Изучена технология CUDA для проведения вычислений на графической карте.

2. Разобраны функции и атрибуты, предоставляемые CUDA.

3. Изменен последовательный код так, чтобы перенести часть вычислений на графический процессор. В результате чего было получено ускорение вычислений примерно в шесть раз.

Список использованных источников

1. Алгоритм многоуровневой адаптации сеток по критериям на основе вейвлет-анализа для задач газовой динамики, А.Л.Афендиков [и др.] Препринты ИПМ им. М.В.Келдыша. 2015. № 97.
2. Богачев К.Ю. Основы параллельного программирования. – М.: БИНОМ. Лаборатория знаний, 2003.
3. Колган В.П. Применение принципа минимальных значений производных к построению конечно-разностных схем для расчета разрывных решений газовой динамики // Ученые записки ЦАГИ, 1972, 3, №6, с. 68-77.
4. Якобовский М.В. Введение в параллельные методы решения задач: Учебное пособие / Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2013. – 328 с., илл. – (Серия «Суперкомпьютерное образование»), список исправлений: http://lira.imamod.ru
5. Кудряшов И.Ю., Луцкий А.Е., Северин А.В. Численное исследование отрывного трансзвукового обтекания моделей с сужением хвостовой части // Препринты ИПМ им. М.В.Келдыша. 2010. № 7. 12 с. URL: http://library.keldysh.ru/preprint.asp?id=2010-7
6. Вычисления на графических процессорах (GPU) в задачах математической и теоретической физики. Перепёлкин Е.Е., Садовников Б.И., Иноземцева.
7. Технология CUDA в примерах. Джейсон Сандерс, Эдвард Кэндрот.
8. Основы работы с технологией CUDA. А. В. Боресков, А. А. Харламов.
9. https://cyberleninka.ru/article/n/ispolzovanie-videokart-dlya-vypolneniya-vychisleniy-pri-reshenii-zadach-stroitelnoy-mehaniki-metodom-konechnyh-elementov/viewer
10. https://cfd.spbstu.ru/agarbaruk/doc/Bach_2016_Fedotov_GPU.pdf