



Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Робототехники и комплексной автоматизации

КАФЕДРА Системы автоматизированного проектирования (РК-6)

ОТЧЕТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНОЙ РАБОТЫ

Студент Лубянов Александр Дмитриевич

Группа РК6-11М

Тип задания лабораторная работа

Тема лабораторной работы Увеличение скорости программы за счёт
распараллеливания средствами OpenMP и CUDA

Студент _____ Лубянов, А. Д.
подпись, дата фамилия, и.о.

Преподаватель _____ Спасёнов, А. Ю.
подпись, дата фамилия, и.о.

Оценка _____

Москва, 2020 г.

Оглавление

Задание на лабораторную работу	3
Цель выполнения лабораторной работы.....	Ошибка! Закладка не определена.
Задачи, выполненные в процессе реализации лабораторной работы	Ошибка! Закладка не определена.
1. Аналитическое решение	Ошибка! Закладка не определена.
2. Построение графовой модели	Ошибка! Закладка не определена.
2.1. Решение прямой задачи	Ошибка! Закладка не определена.
2.2. Решение обратной задачи.....	Ошибка! Закладка не определена.
2.3. Задание исходных данных.....	Ошибка! Закладка не определена.
Заключение	12
Список использованных источников	13

Задание на лабораторную работу

Усовершенствовать программу для лабораторной работы по МКР с помощью средств библиотеки OpenMP и технологии CUDA для ускорения работы.

Задание для лабораторной работы по МКР:

С помощью неявной разностной схемы решить нестационарное уравнение теплопроводности для трубы, изображенной на рисунке 1, там также указаны размеры сторон.

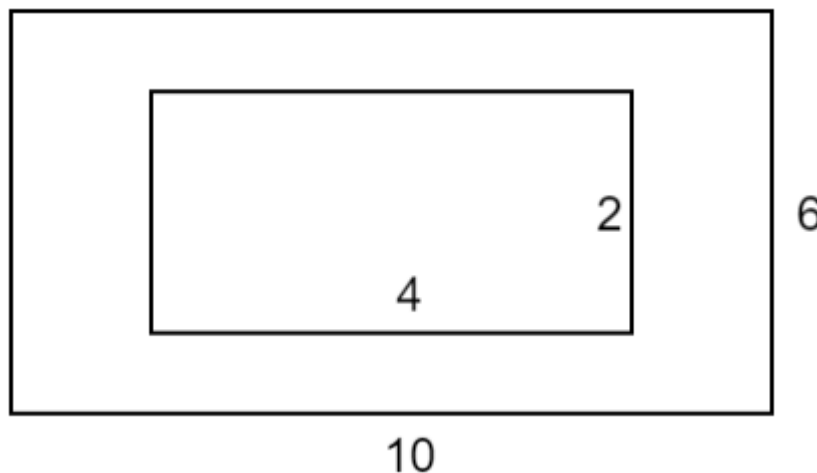


Рисунок 1. Сечение трубы

Граничные условия: внутри трубы протекает жидкость температурой 100 градусов, на внешней границе задано условие:

$$dT/dn = T.$$

Начальное значение температуры трубы – 10 градусов.

Результат работы, визуализация объекта с распределением температуры в момент времени, 10 сек показан на рисунке 2.

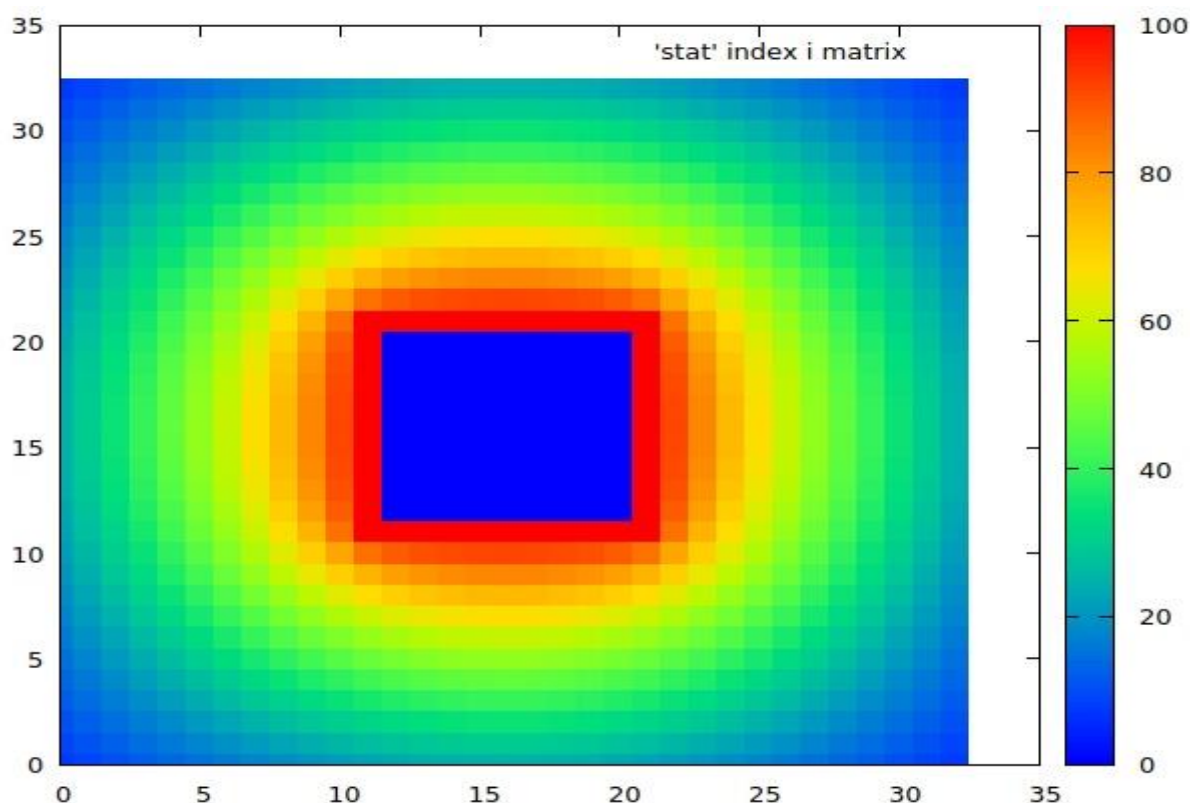


Рисунок 2. Результат работы программы

1. Реализация с помощью OpenMP

1.1. Описание библиотеки OpenMP

OpenMP (Open Multi-Processing) — открытый стандарт для распараллеливания программ на языках Си, Си++ и Фортран. Дает описание совокупности директив компилятора, библиотечных процедур и переменных окружения, которые предназначены для программирования многопоточных приложений на многопроцессорных системах с общей памятью.

OpenMP реализует параллельные вычисления с помощью многопоточности, в которой «главный» (master) поток создает набор подчиненных (slave) потоков, и задача распределяется между ними. Предполагается, что потоки выполняются параллельно на машине с несколькими процессорами (количество процессоров не обязательно должно быть больше или равно количеству потоков). Задачи, выполняемые потоками параллельно, так же, как и данные, требуемые для выполнения этих задач, описываются с помощью специальных директив препроцессора соответствующего языка — прагм [1].

1.2. Применение OpenMP

В листинге 1 содержится часть текста программы, в которой были использованы средства данной библиотеки. Т.к. основное время работы программы занимало решение системы уравнений методом Гаусса, именно эта часть подверглась изменениям.

При компиляции программы использовался флаг `-fopenmp`.

Листинг 1 – Решение СЛАУ методом Гаусса с помощью OpenMP

```
1.      // прямой ход Гаусса
2.      #pragma omp parallel for default(shared)
3.      for(int j = 0; j < n; j++)
4.      {
5.          for (int i = j + 1; i < n; i++)
6.          {
7.              double koef = matr[i][j] / matr[j][j];
8.              for(int k = j; k < n; k++)
9.              {
10.                 matr[i][k] -= koef * matr[j][k];
11.             }
12.
13.             slv[i] -= koef * slv[j];
14.         }
15.     }
16.
17.     // обратный ход Гаусса
18.     #pragma omp parallel for default(shared)
19.     for(int i = n - 1; i >= 0; i--)
20.     {
21.         double sum = 0.0;
22.         for(int j = i + 1; j < n; j++)
23.         {
24.             sum += solve[j] * matr[i][j];
25.         }
26.
27.         solve[i] = (slv[i] - sum) / matr[i][i];
28.     }
```

1.3. Результаты работы

Программа тестировалась на компьютере, обладающим 4-х ядерным процессором с 8 потоками. Время работы при выставленных настройках в 10 и 20 секунд показано в таблице 1.

Таблица 1. Время выполнения программы с OpenMP при различных входных данных

Заданное время, сек	Количество потоков				
	1	2	4	8	16
10	20.7 с	12.6 с	8.6 с	7.2 с	8.2 с
20	34.4 с	20.6 с	13.3 с	11.9 с	12.9 с

Как видно из таблицы 1, наилучшее время исполнения программа показывает при 8 потоках (столько же, сколько в процессоре). Дальнейшее увеличения количества потоков приводит к замедлению из-за траты времени на распараллеливание. При этом увеличение производительности при переходе с 4 на 8 потоков заметно падает по сравнению с предыдущими из-за того, что количество ядер процессора равно 4. Также стоит отметить, что часть времени тратится на последовательные вычисления.

2. Реализация с помощью CUDA

2.1. Описание технологии NVIDIA CUDA

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

CUDA SDK позволяет программистам реализовывать на специальных упрощённых диалектах языков программирования Си, С++ и Фортран алгоритмы, выполнимые на графических и тензорных процессорах Nvidia. Архитектура CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций графического или тензорного ускорителя и управлять его памятью [2].

Выполнение расчётов на GPU показывает отличные результаты в алгоритмах, использующих параллельную обработку данных. То есть, когда одну и ту же последовательность математических операций применяют к большому объёму данных. При этом лучшие результаты достигаются, если отношение числа арифметических инструкций к числу обращений к памяти достаточно велико. Это предъявляет меньшие требования к управлению исполнением (flow control), а высокая плотность математики и большой объём данных отменяет необходимость в больших кэшах, как на CPU.

Основные характеристики CUDA:

- унифицированное программно-аппаратное решение для параллельных вычислений на видеочипах Nvidia;
- большой набор поддерживаемых решений, от мобильных до мультичиповых
- стандартный язык программирования Си;
- стандартные библиотеки численного анализа FFT (быстрое преобразование Фурье) и BLAS (линейная алгебра);
- оптимизированный обмен данными между CPU и GPU;
- взаимодействие с графическими API OpenGL и DirectX;
- поддержка 32- и 64-битных операционных систем: Windows XP, Windows Vista, Linux и MacOS X;
- возможность разработки на низком уровне.

Преимущества CUDA перед традиционным подходом к GPGPU вычислениям:

- интерфейс программирования приложений CUDA основан на стандартном языке программирования Си с расширениями, что упрощает процесс изучения и внедрения архитектуры CUDA;
- CUDA обеспечивает доступ к разделяемой между потоками памяти размером в 16 Кб на мультипроцессор, которая может быть использована для организации кэша с широкой полосой пропускания, по сравнению с текстурными выборками;
- более эффективная передача данных между системной и видеопамятью
- отсутствие необходимости в графических API с избыточностью и накладными расходами;
- линейная адресация памяти, и gather и scatter, возможность записи по произвольным адресам;
- аппаратная поддержка целочисленных и битовых операций.

- Основные ограничения CUDA:
- отсутствие поддержки рекурсии для выполняемых функций;
- минимальная ширина блока в 32 потока;
- закрытая архитектура CUDA, принадлежащая Nvidia [3].

2.2. Применение CUDA

Для компиляции программы с расширением .cu был установлен компилятор для кода CUDA. Вычисления производились с помощью видеокарты NVIDIA GeForce GTX 1060 with Max-Q Design.

В ходе работы были разработаны 2 функции, выполняющие прямой и обратный ход Гаусса (листинг 2). Они имеют спецификатор `__global__`, который показывает, что функция вызывается с хоста и выполняется на устройстве.

Листинг 2 – функции, выполняемые на видеокарте

```

1. __global__ void Gauss_forward(double *matr, double *slv, int n)
2. {
3.     int j = blockIdx.y * blockDim.y + threadIdx.y;
4.     for (int i = j + 1; i < n; i++)
5.     {
6.         double koef = matr[i*N + j] / matr[j*N + j];
7.         for(int k = j; k < n; k++)
8.         {
9.             matr[i*N + k] -= koef * matr[j*N + k];
10.        }
11.
12.        slv[i] -= koef * slv[j];
13.    }
14.}
15.
16.
17. __global__ void Gauss_reverse(double *matr, double *slv, double *solve, int n)
18. {
19.     int k = n - 1 - blockIdx.x * blockDim.x + threadIdx.x;
20.     double sum = 0.0;
21.     for(int j = k + 1; j < n; j++)
22.     {
23.         sum += solve[j] * matr[k*N + j];
24.     }
25.
26.     solve[k] = (slv[k] - sum) / matr[k*N + k];
27.}

```


Данные функции вызываются из главной функции *int main()*, в которой находится основная логика программы. Фрагмент её текста представлен в листинге 3. Для правильной работы с технологией CUDA были использованы следующие функции:

- `cudaMalloc` – для выделения памяти под массивы CUDA;
- `cudaEventCreate` – для создания event'a;
- `cudaMemcpy` – для передачи данных из массива хоста в массив девайса и наоборот;
- `cudaDeviceSynchronize` – для синхронизации потоков;
- `cudaGetLastError` – для проверки на ошибки;
- `cudaFree` – для освобождения памяти.

Листинг 3 – Фрагмент функции `int main()`

```
1.      int n = N;
2.      double *device_matr;
3.      double *device_slv;
4.      double *device_solve;
5.      unsigned int size_matr = sizeof(double) * n * n;
6.      unsigned int size_slv = sizeof(double) * n;
7.
8.      cudaError cudaStatus;
9.
10.     cudaMalloc((void**)&device_matr, size_matr);
11.     cudaMalloc((void**)&device_slv, size_slv);
12.     cudaMalloc((void**)&device_solve, size_slv);
13.
14.     float timerValueGPU;
15.     cudaEvent_t start2, stop2;
16.     cudaEventCreate(&start2);
17.     cudaEventCreate(&stop2);
18.     cudaEventRecord(start2, 0);
19.
20.     dim3 N_Treads(8);
21.     dim3 N_Block(n / 8);
22.
23.     for (int k = 0; k <= TIME / h_t; k+= h_t)
24.     {
25.         cout << k << endl;
26.         for (int j = 1; j < X; ++j)
27.         {
28.             solve[j] = 0;
29.             solve[j + (X + 1) * Y] = 0;
30.         }
```

```

31.
32.     for (int i = 0; i <= Y; ++i)
33.     {
34.         solve[i * (X + 1)] = 0;
35.         solve[i * (X + 1) + X] = 0;
36.     }
37.
38.     cudaMemcpy(device_solve, solve, size_slv, cudaMemcpyHostToDevice);
39.
40.     double *matr2;
41.     matr2 = (double*)malloc(sizeof(double) * N * N);
42.
43.     double *slv = new double[n];
44.
45.     for (int i = 0; i < N; i++)
46.     {
47.         for (int j = 0; j < N; j++)
48.         {
49.             matr2[i*N + j] = matr[i*N + j];
50.         }
51.         slv[i] = solve[i];
52.     }
53.
54.     cudaMemcpy(device_matr, matr2, size_matr, cudaMemcpyHostToDevice);
55.
56.     cudaMemcpy(device_slv, slv, size_slv, cudaMemcpyHostToDevice);
57.
58.     Gauss_forward <<< N_Block, N_Treads >>> (device_matr, device_slv, n);
59.
60.     cudaStatus = cudaGetLastError();
61.     if (cudaStatus != cudaSuccess)
62.     {
63.         cout << "Solve last error:" << cudaGetErrorString(cudaStatus)
64.         << endl;
65.         return 0;
66.     }
67.     cudaDeviceSynchronize();
68.
69.     cudaMemcpy(slv, device_slv, size_slv, cudaMemcpyDeviceToHost);
70.     cudaMemcpy(matr2, device_matr, size_matr, cudaMemcpyDeviceToHost);
71.
72.     Gauss_reverse <<< N_Block, N_Treads >>> (device_matr, device_slv,
73.     device_solve, n);
74.     cudaStatus = cudaGetLastError();
75.     if (cudaStatus != cudaSuccess)
76.     {
77.         cout << "Solve last error:" << cudaGetErrorString(cudaStatus)
78.         << endl;
79.         return 0;
80.     }

```

```

76.         cudaDeviceSynchronize();
77.
78.         cudaMemcpy(solve, device_solve, size_slv, cudaMemcpyDeviceToHost);
79.         cudaMemcpy(slv, device_slv, size_slv, cudaMemcpyDeviceToHost);
80.         cudaMemcpy(matr2, device_matr, size_matr, cudaMemcpyDeviceToHost);
81.
82.         delete matr2;
83.         delete slv;
84.     }
85.
86.     cudaEventRecord(stop2, 0);
87.     cudaDeviceSynchronize();
88.     cudaEventSynchronize(stop2);
89.     cudaEventElapsedTime(&timerValueGPU, start2, stop2);
90.     cout << "GPU calculation time " << timerValueGPU << " msec" << endl;
91.
92.     cudaFree(device_matr);
93.     cudaFree(device_slv);
94.     cudaFree(device_solve);

```

2.3. Результаты работы

В таблице 2 приведена временная оценка работы программы при разных количествах потоков при заданном интервале 10 сек в случае с обычным размером пластины и увеличенным в 1.5 раза.

Таблица 2. Время выполнения программы с CUDA при различных входных данных

Размер пластины	Количество потоков						
	1	2	4	8	16	32	64
стандартный	14.0 с	7.2 с	3.7 с	3.4 с	3.3 с	3.2 с	3.3 с
увеличенный в 1.5 раза	134.2 с	65.7 с	34.3 с	17.8 с	15.4 с	15.3 с	15.2 с

Как видно из таблицы, при увеличении количества потоков программа работает быстрее, но при определённом значении потоков рост скорости уменьшается, а затем и вовсе прекращается. При этом для разных размеров матриц значения оптимального количества потоков (для обычного размера – 4, для увеличенного – 8) отличается. Значит, при разработке программы с

параллельным вычислением следует подбирать количество потоков в зависимости от размера обрабатываемых данных. Чем они больше – тем больше потоков будет оптимально использовать.

Заключение

В ходе данной лабораторной работы была распараллелена программа для расчёта температуры при помощи МКР средствами библиотеки OpenMP и технологии NVIDIA CUDA. Изучены основные функции, подходы параллельного вычисления, полученные знания применены при разработке. Получены результаты времени работы программы при разных входных данных и разном количестве потоков. Проанализированы полученные результаты. Для заданной размерности задачи можно подобрать оптимальное количество параллельных процессов, при котором эффективность распараллеливания будет наибольшей.

Список использованных источников

[1] OpenMP [Электронный ресурс]. Википедия [Официальный сайт]. (дата обращения 20.12.2020) URL: <https://ru.wikipedia.org/wiki/OpenMP>

[2] CUDA [Электронный ресурс]. Википедия [Официальный сайт]. (дата обращения 21.12.2020) URL: <https://ru.wikipedia.org/wiki/CUDA>

[3] Nvidia CUDA. Неграфические вычисления на графических процессорах [Электронный ресурс]. IXBT (дата обращения 21.12.2020) URL: <https://www.ixbt.com/video3/cuda-1.shtml>