КАРЧИГАНОВ А. Ф.

ИСПОЛЬЗОВАНИЕ ТЕХНОЛОГИИ OPENCL ДЛЯ ВЫЧИСЛЕНИЙ НА СТРУКТУРИРОВАННЫХ СЕТКАХ С ИСПОЛЬЗОВАНИЕМ GPU

Аннотация. В статье рассматривается возможность применения технологии OpenCL для параллельного вычисления численного решения на примере двумерной задачи теплопроводности. Рассмотрены особенности программирования в парадигме параллелизма задач и данных. Показаны стандартные сложности и встроенные в OpenCL способы их решения при построении программы для вычислений на многомерных сетках. Для CPU реализована аналогичная однопоточная программа на языке C++20, сравнена производительность.

Ключевые слова: OpenCL, гетерогенные вычисления, ОКМД, уравнение теплопроводности.

KARCHIGANOV A. F.

USING OPENCL TECHNOLOGY FOR STRUCTURED GRID COMPUTING USING GPU

Abstract. In this article the possibility of using OpenCL technology for parallel calculation of a numerical solution using the example of a two-dimensional heat conduction problem is discussed. The features of programming in the paradigm of task and data parallelism are considered. The standard difficulties and methods built into OpenCL for solving them when building a program for calculations on multidimensional grids are shown. For the CPU, a similar single-threaded program was implemented in C++20, and the performance was compared.

Keywords: OpenCL, heterogeneous computing, SIMD, heat equation.

Появление многоядерных процессоров и массовый их выход в потребление, сделал программирование под суперкомпьютеры доступным для широкого круга. Одно из основных применений суперкомпьютера – численное моделирование, сопряженное с очень большим объёмом сложных вычислений.

Другая веха в развитии численного моделирования – появление возможности использования графических процессоров для неграфических вычислений. Графические процессоры, с одной стороны, имея меньший набор команд по сравнению с процессорами общего назначения, с другой – имеют намного большее число ядер, что, с определенного этапа их развития, сделало целесообразным их использование для математических вычислений.

Одна из первых и наиболее широко используемых технологий остается CUDA [1]. Имея достаточно простой интерфейс взаимодействия под языки C/C++, а также реализованные

обертки для других языков программирования, CUDA имеет и свой основной недостаток – возможность запуска параллельных программ только под видеокарты компании Nvidia.

Появление стандарта для открытого языка вычислений OpenCL сделало доступным возможность написания общих многопоточных программ не только под видеокарты, но и под другие аппаратные ускорители, в том числе и процессоры общего назначения. Имея похожий на CUDA интерфейс взаимодействия через C/C++ и другие языки программирования, для самих математических вычислений OpenCL использует подход близкий к языку шейдеров OpenGL SL, который так же использовал программы на собственном языке, близком к C; а выходные буферы могли быть использованы не только при выводе изображения на экран пользователя, но и для прямого копирования данных с них в общую оперативную память компьютера, чем также пользовались [2].

Целью данной работы было исследование возможности использования технологии OpenCL для решения задач на структурированных сетках, выявление типовых сложностей при разработке программ на принципах параллелизма данных.

В качестве примера выбрана двумерная задача теплопроводности для однородного тела с начальными и краевыми условиями [3]: однородная пластина с коэффициентом теплопроводности λ, шириной W и высотой H, имеет начальную температуру в момент времени $t = 0$, заданную функцией $T_0(x, y)$, где $x \in [0; W]$, $y \in [0; H]$. Грани пластины также могут быть подвержены внешнему воздействию, т.е. на них могут быть заданы функции $T_u(t, x),\ T_d(t, x),\ T_l(t, y),\ T_r(t, y)$ для соответственно верхней, нижней, левой, правой границ. Для подобных задач можно построить явную конечно-разностную схему [3].

Разбив область $x \times y$ на $N_x \times N_y$ ячеек с соответствующими шагами сетки $h_x \times h_y$, установив шаг по времени τ, выразим значение в ячейке i, j для каждого следующего момента времени:

$$T_{i,j}^{+} = T_{i,j} + \tau\lambda\left(\frac{\left(T_{i-1,j} + T_{i+1,j} - 2T_{i,j}\right)}{h_x^2} + \frac{\left(T_{i,j-1} + T_{i,j+1} - 2T_{i,j}\right)}{h_y^2}\right), \quad (1)$$

где $i = 1, \dots, N_x,\ j = 1, \dots, N_y$. Для удобства, будем использовать дополнительные слои при $i \in [0, N_x + 1],\ j \in \left[0, N_y + 1\right]$ для задания значений на левых, правых, верхних и нижних границах соответственно. Тогда полученная схема с ячейками $[j, i]$ будет выглядеть:

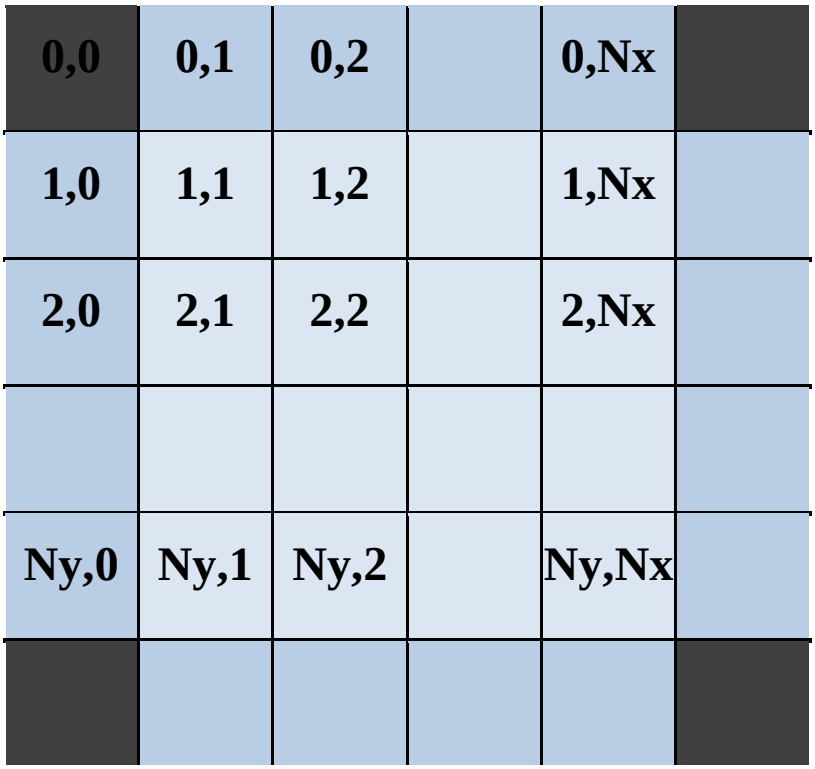

Рис. 1. Разбиение области на ячейки.

где светло-голубым цветом выделены ячейки, рассчитываемые по формуле (1), темно-голубым – через граничные условия, черные – полностью неиспользуемые в программном алгоритме.

Была реализована host-программа, которая связывается с устройствами, поддерживающими драйвер OpenCL, инициализирует начальные данные, собирает kernel-программу, заносит и вытаскивает данные из буферов графической памяти. При составлении host- и kernel-программ необходимо учитывать некоторые особенности работы графического процессора.

Так как в графическом процессоре для достижения больше производительности используются обычно числа с плавающей точкой одинарной точности (32-битные), то и при создании данных на host-программе необходимо использовать их, а не 64-битные. В случае с C++20 – это `float_t`.

При работе с памятью графического ускорителя, невозможно случайным образом занимать и освобождать память в глобальной области, то есть все указатели на массивы данных должны быть известны заранее, а значит невозможно загрузить внутрь графической памяти двумерный массив как указатель на указатели. Соответственно, область из рисунка 1 необходимо представить в виде одномерного массива. Заполним структуру `std::<float_t>`, редуцировав двумерный массив до одномерного:

```
auto valuesArray = std::vector<float_t>(fullSize);
for (size_t j = 0; j < height.Count; ++j)
{
	auto heightOffset = j * width.Count;
	for (size_t i = 0; i < width.Count; ++i)
	{
		valuesArray[heightOffset + i] = initialFunciton(
			width.Coordinates[i],
			height.Coordinates[j]
		);
	}
}
```

Листинг 1. Инициализация начальных данных.

где `initialFunciton` – функция $T_0(x, y)$.

Полученный массив из-за граничных ячеек имеет $\left((N_x + 2) \cdot \left(N_y + 2\right)\right)$ элементов (до Листинга 1 вычислено в переменной `fullSize`), в то время как программа для вычисления по формуле (1) должна быть применена только на $\left(N_x \cdot N_y\right)$ элементов.

Программы на OpenCL пишутся в парадигме параллелизма, то есть необходимо описывать не циклы (как в Листинге 1), а функции, которые будут отрабатывать на каждой рабочей единице. Например, программа для сложения массивов `A` и `B` в массив `C` размерностью 32 будет выглядеть:

```
__kernel void add(__global float *A, __global float *B, __global float *C)
{
	size_t i = get_global_id(0);
	C[i] = A[i] + B[i];
}
```

Листинг 2. Простейший алгоритм сложения массивов.

где `get_global_id(0)` – получение глобального номера рабочей единицы в первом измерении. В случае, когда вычисления на граничных ячейках нужно пропустить, нельзя использовать следующий код:

```
__kernel void add(__global float *A, __global float *B, __global float *C)
{
	size_t i = get_global_id(0);
	if ((i > 0) && (i <= 32)
	{
		C[i] = A[i] + B[i];
	}
	else
	{ … }
}
```

Листинг 3. Пример дивергенции кода.

так как произойдет дивергенция кода – ситуация, когда в целях синхронизации все рабочие единицы затратят время для отработки или ожидания сначала блока `if`, а потом и блока `else`.

Необходимо сделать отступы для соответствующих измерений. Подобная задача для графических ускорителей встречалась и ранее: например, задача Гауссова размытия [4]. OpenCL позволяет задавать размерности редуцированного массива, а также отступы от начала для каждой размерности [5]. Таким образом, для обоих размерностей необходимо сделать отступы размером в 2, а расчетная область будет выглядеть следующим образом:

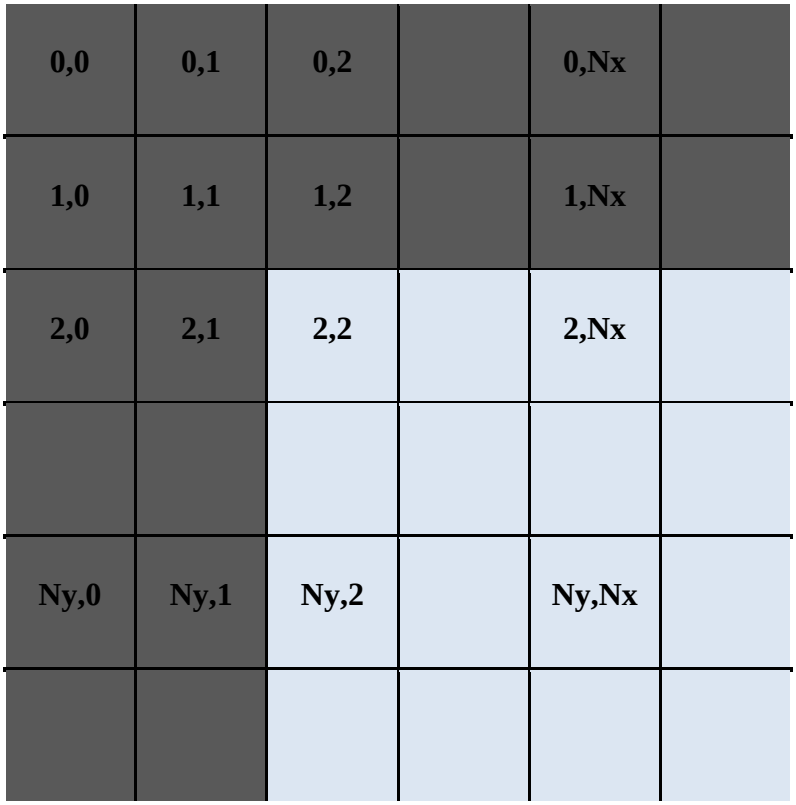

Рис. 2. Рабочее пространство для kernel-программы.

где светло-голубым цветом обозначены ячейки, относительно которых будут распределяться индексы рабочих единиц. Тогда сами индексы $[i, j]$ для соответствия формуле (1) можно будет определить по соответствующим измерениям:

```
int i = get_global_id(0) - 1;
int j = get_global_id(1) - 1;
```

Листинг 4. Определение индексов вычисляемой ячейки.

В host-программе создадим текст kernel-программы, реализующей вычисление по формуле (1):

```
	auto programTextTemplate = R"(
__kernel void conduction(
	float coefficientX, float coefficientY,
	__global float *input, __global float *output
)
{{
	int index0 = get_global_id(0) - 1;
	int index1 = get_global_id(1) - 1;
	int index = (index1 * {0}) + index0;
	output[index] = input[index]
		+ (coefficientX * (
			input[index - 1] + input[index + 1] - (2 * input[index])
		))
		+ (coefficientY * (
			input[index - {0}] + input[index + {0}] - (2 * input[index])
		));
}}
)";
	auto programTextTemplateArgs = std::make_format_args(width.Count);

	auto programText = std::vformat(programTextTemplate, programTextTemplateArgs);
```

Листинг 5. Составление текста kernel-программы.

где в случае, когда в переменной `width.Count` будет находиться значение $N_x = 1024$, в переменной `programText` будет составлен следующий текст kernel-программы:

```
__kernel void conduction(
	float coefficientX, float coefficientY,
	__global float *input, __global float *output
)
{
	int index0 = get_global_id(0) - 1;
	int index1 = get_global_id(1) - 1;
	int index = (index1 * 1024) + index0;
	output[index] = input[index]
		+ (coefficientX * (
			input[index - 1] + input[index + 1] - (2 * input[index])
		))
		+ (coefficientY * (
			input[index - 1024] + input[index + 1024] - (2 * input[index])
		));
}
```

Листинг 6. Пример kernel-программы при $N_x = 1024$.

где `coefficient` и `coefficientY` вычислены заранее как $\frac{\tau\lambda}{h_x^2}$ и $\frac{\tau\lambda}{h_y^2}$ из формулы (1) соответственно.

Так как задачи в OpenCL выполняются асинхронно, то в них присутствуют очереди. Все задачи в одной очереди выполняются друг за другом. Различные очереди выполняются

параллельно. При создании задачи, host-программа автоматически не ждет ее выполнения, но имеет есть возможность ожидания задачи для синхронизации. Составим очередь задач, где данные будут загружаться в память GPU, выполняться по программе из Листинга 6, и выгружаться обратно в общую оперативную память:

```
auto previousEvents = std::vector<cl::Event>(0);
auto writeEvent = cl::Event();
errorCode = commandQueue.enqueueWriteBuffer(
      inputBuffer, CL_TRUE, 0, fullSize * sizeof(float_t), valuesArray.data(),
      &previousEvents, &writeEvent
);
previousEvents.push_back(writeEvent);
auto kernelEvent = cl::Event();
errorCode = commandQueue.enqueueNDRangeKernel(
      kernel, ndOffset, ndGlobal, cl::NullRange,
      &previousEvents, &kernelEvent
);
previousEvents.push_back(kernelEvent);
auto readEvent = cl::Event();
errorCode = commandQueue.enqueueReadBuffer(
      outputBuffer, CL_TRUE, 0, fullSize * sizeof(float_t), valuesArray.data(),
      &previousEvents, &readEvent
);
readEvent.wait();

errorCode = commandQueue.finish();
```

Листинг 7. Очередь задач.

Данный алгоритм выполняется на каждом шаге по времени. Между его выполнениями, на host-программе вычисляются значения для граничных ячеек по функциям T_u, T_d, T_l, T_r.

В соответствии с (1) была также составлена аналогичная однопоточная программа на C++20. Программы на OpenCL и на C++20 были сравнены по производительности: на сетке размерностью 1024×1024 с числом шагов $10'000$ (максимальное время $T = 1.0$, шаг по времени $\tau = 0.0001$) программа на GPU отработала за 15 секунд, на CPU – 610 секунд. Программу на OpenCL можно еще ускорить, реализовав программы для функций T_u, T_d, T_l, T_r, тем самым исключив время для загрузки и выгрузки данных с графической памяти на каждом шаге. Полученные значения массивов совпали побитово.

Вычисления, отдаваемые устройствам под управлением OpenCL, выполняются на всех вычислительных единицах этого устройства. OpenCL имеет поддержку деления устройства только для некоторых процессоров, поэтому проверить эффективность и ускорение алгоритма на ограниченном количестве ядер графического процессора невозможно.

Программы были также проверены при $T = 2.0$, $\tau = 0.0001$ на разном количестве ячеек. Вычислена производительность и эффективность относительно вычисления с предыдущим количеством ячеек (таблица 1).

Таблица 1

Производительность и эффективность работы программы

Время, мс	$N_x \times N_y$	$N_x N_y$	**Производительность, ячеек/мс**	**Эффективность**
5452	256×256	65536	12.02054	
10005	256×512	131072	13.10065	1.089855
18224	512×512	262144	14.38455	1.098003
32202	512×1024	524288	16.28122	1.131855
60771	1024×1024	1048576	17.25455	1.059782

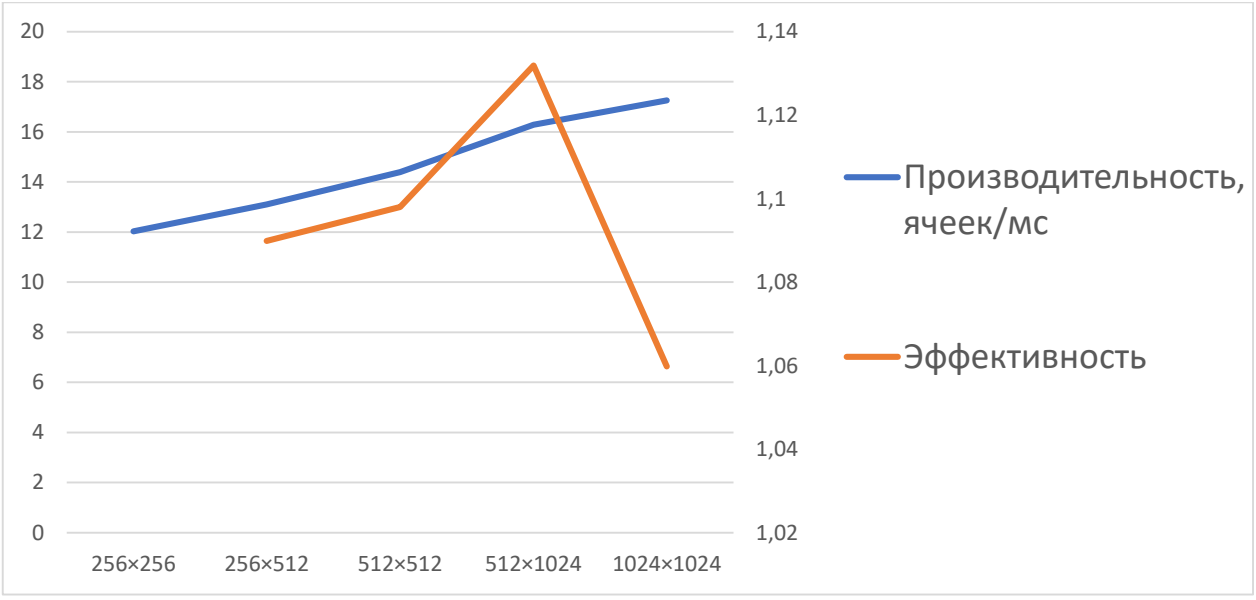

Рис. 3. Производительность и эффективность работы программы.

В ходе выполнения данной работы были изучены особенности написания и работы программ с использованием OpenCL. Была реализована host-программа на языке C++20 для составления и запуска kernel-программы на OpenCL. Производительность и эффективность работы программы на OpenCL были замерены, а также была сравнена производительность с аналогичной однопоточной программой под процессор общего назначения. На основании полученных результатов можно сделать вывод, что программы на OpenCL имеют существенно высокую производительность, а также хорошую эффективность на больших объемах данных.

СПИСОК ЛИТЕРАТУРЫ

1. Антонюк В. А. Программирование на видеокартах (GPGPU). Спецкурс кафедры ММИ. – М.: Физический факультет МГУ им. М.В. Ломоносова, 2015. – 48 с.
2. Вычисления на GPU с помощью OpenGL [Электронный ресурс]. – Режим доступа: https://velikodniy.github.io/2017/08/14/gpgpu-opengles (дата обращения: 22.10.2023).
3. Кузнецов Г. В., Шеремет М. А. Разностные методы решения задач теплопроводности: учебное пособие. – Томск: Изд-во ТПУ, 2007. – 172 с.
4. Compute shaders in graphics: Gaussian blur [Электронный ресурс]. – Режим доступа: https://lisyarus.github.io/blog/graphics/2022/04/21/compute-blur.html (дата обращения: 22.10.2023).
5. Erik S. Gaussian Blur using OpenCL and the built-in Images/Textures [Электронный ресурс]. – Режим доступа: https://www.eriksmistad.no/gaussian-blur-using-opencl-and-the-built-in-images-textures (дата обращения: 22.10.2023).