Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Архитектура вычислительных систем

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«Быстрое преобразование Фурье и его распараллеливание (под архитектуру CUDA для NVIDIA)»

БГУИР КП 1-40 04 01 009 ПЗ

Студентка гр. 753504
Борисова А. С.
Руководитель
ассистент кафедры информатики
Леченко А. В.

Минск 2019

Содержание

1. Быстрое преобразование Фурье

Быстрое преобразование Фурье (БПФ, FFT) — алгоритм для более эффективного вычисления дискретного преобразования Фурье (ДПФ, DFT), которое напрямую вычисляется за $O(N^2)$. Большинство алгоритмов вычисления БПФ базируется на алгоритме Кули-Тьюки.

В данной работе рассматривалось применение БПФ в умножении длинных чисел, что схоже с умножением многочленов.

В таком случае ДПФ представляет собой вычисление значений многочлена в комплексных корнях из единицы:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-i2\pi kn}{n}}$$

ОДПФ же представляет собой интерполяцию коэффициентов x_i по значениям X_i:

$$x_n = \frac{1}{N} \cdot \sum_{k=0}^{N-1} X_k \cdot e^{\frac{-i2\pi kn}{n}}$$

Таким образом, можно отметить сходство вычислений прямого и обратного преобразования, благодаря которому для вычисления ОДПФ можно использовать тот же алгоритм с минимальными поправками.

Перемножение многочленов $A(x)$ и $B(x)$ будет производиться следующим образом:

$$(A \cdot B)(x) = A(x) \cdot B(x)$$
$$DFT(A \cdot B) = DFT(A) \cdot DFT(B)$$
$$A \cdot B = InverseDFT(DFT(A) \cdot DFT(B))$$

При вычислении ОДПФ производится произведение ДПФ A и B, что представляет собой попарные произведения элементов векторов. Такое произведение требует для вычисления $O(N)$ операций. Следовательно, если вычислять ДПФ и обратное ДПФ за время $O(N\ logN)$, используя алгоритм БПФ, то произведение двух полиномов (как двух длинных чисел) также вычисляется с такой асимптотикой.

Используемый алгоритм Кули-Тьюки вычисления БПФ базируется на использовании симметрии в формулах: вычисления БПФ можно разделить на две части, разделяя исходный многочлен на два многочлена — с четными и нечетными коэффициентами:

$$A(x) = A_0(x^2) + x \cdot A_1(x^2)$$
$$w_n^k = e^{\frac{-i2\pi kn}{n}}$$

$$w_n^n = e^{-i2\pi} = cos2\pi + isin2\pi = 1$$
$$w_n^{n/2} = e^{-i\pi} = cos\pi + isin\pi = -1$$
$$X_k = A_0(w_n^2 k) + w_n^k \cdot A_1(w_n^2 k) = x_k^0 + w_n^k \cdot x_k^1$$

2. Реализация разных подходов к вычислению преобразования Фурье

2.1. Наивная реализация.

Наивная реализация предполагает использование рекурсии для разделения вектора коэффициентов на векторы с четными и нечетными индексами, для которых затем рекурсивно вычисляется БПФ.

2.2. Оптимизированная реализация.

Для оптимизации требуется отказаться от рекурсии путем переупорядочивания элементов в массиве, что также позволяет добиться разделения вектора коэффициентов на части с четными и нечетными индексами коэффициентов, но вычисления производятся уже «на месте».

2.3. Реализация с использованием OpenMP.

Данная реализация ускоряет уже оптимизированный алгоритм вычисления путем использования OpenMP для параллельных вычислений на CPU.

2.4. Реализация с использованием CUDA.

Реализация с использованием CUDA предполагает использование cuFFT, the CUDA Fast Fourier Transform Library, которая обеспечивает быстрое вычисление БПФ на NVIDIA GPUs. Алгоритм преобразования обеспечивает асимптотику $O(N \log N)$.

3. Результаты тестирования алгоритмов

Для анализа эффективности работы алгоритмов были проведены замеры времени работы при разных объемах входных данных.

Ступенчатость графиков зависимости времени работы от входных данных для БПФ объясняется тем, что, исходя из сути алгоритма, его работа производится с блоками данных, равных степени двойки, и все данные, которые не достигают этого значения, выравниваются. Поэтому большие скачки видны именно на *n*, близких к степеням двойки.

Полный график результатов тестирования:

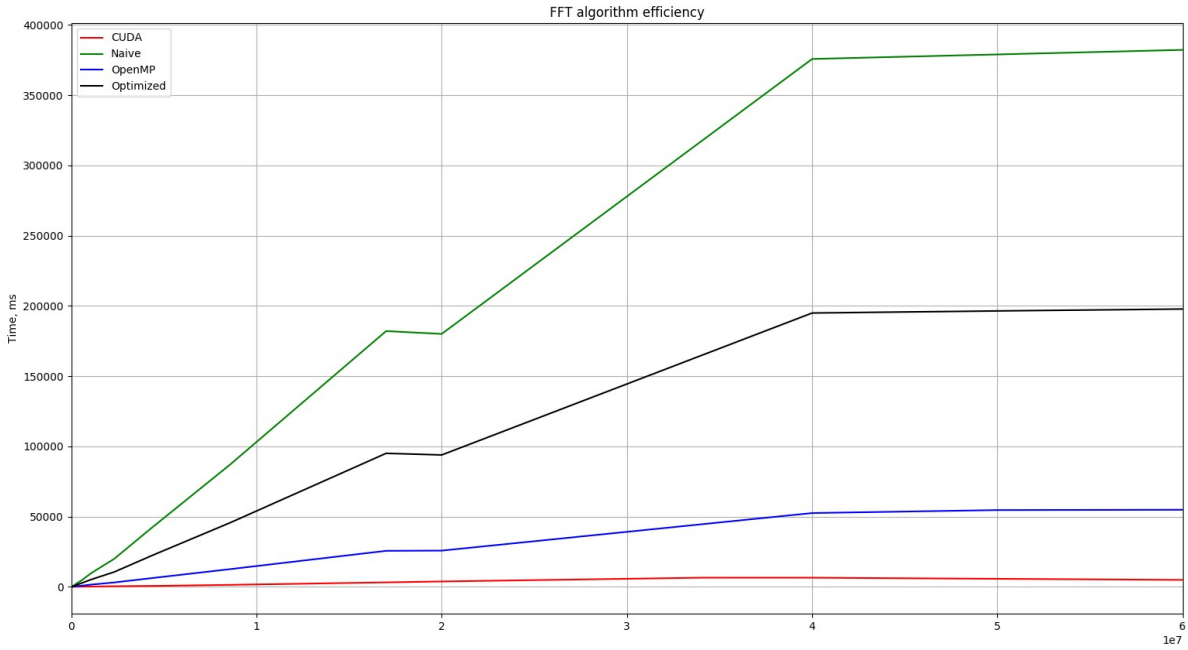

Рис 1. – Полный график результатов тестирования.

3.1. Тестирование на маленьких объемах данных.

Под маленькими объемами данных имеются в виду данные, помещающиеся в кэш компьютера. Кэш ноутбука, на котором проводилось тестирование алгоритмов, имеет объем 6 Мб, что соответствует 2 целочисленным массивам, содержащим 1572864/2 элементов.

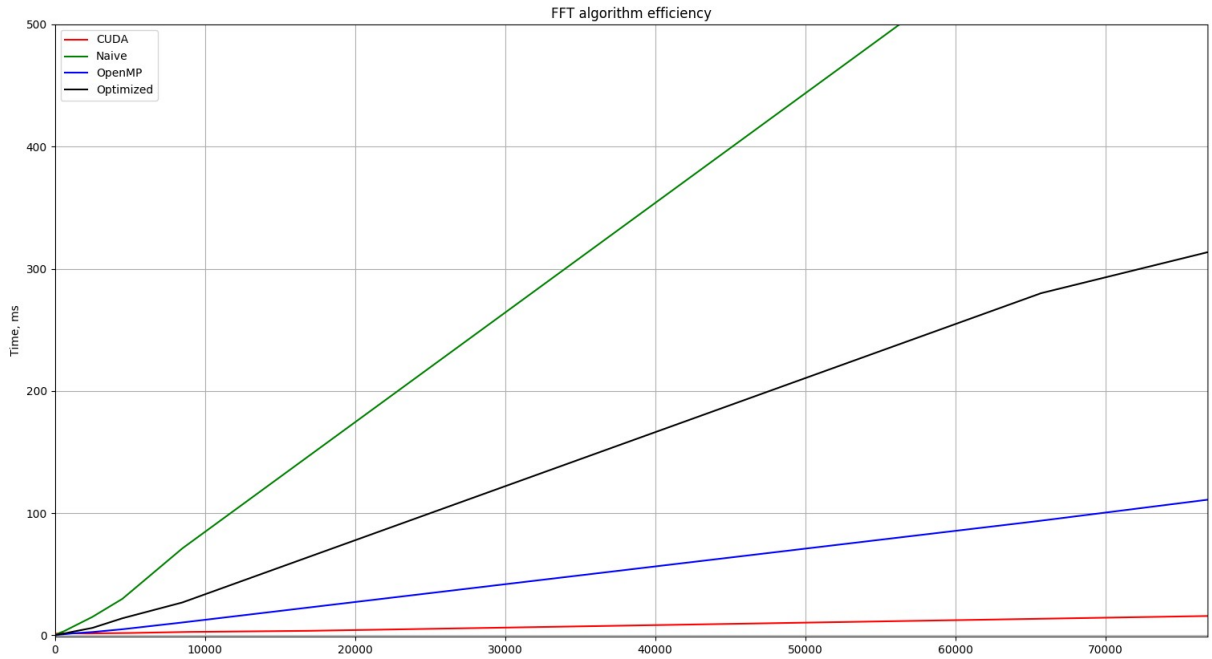

Рис 2. – График результатов тестирования на маленьких объемах данных.

В таблице на рисунке 2 также приводятся среднее время вычисления алгоритма при входных данных, помещающихся в кэш (первая строка), и отношение времени выполнения алгоритма к времени вычислений реализации с использованием CUDA.

Naive	Optimized	OpenMP	CUDA
29.0202	13.12458125	5.33124693877551	2.0589340425531915
14.094769137924086	6.374454440378671	2.589323809598325	1

Рис. 3 – Таблица сравнения результатов тестирования на маленьких объемах данных.

Уже очевидно, что вычисления на GPU существенно быстрее всех прочих.

Также можно отдельно проанализировать результаты тестирования на очень маленьких объемах данных (до 250 элементов в каждом из целочисленных массивов):

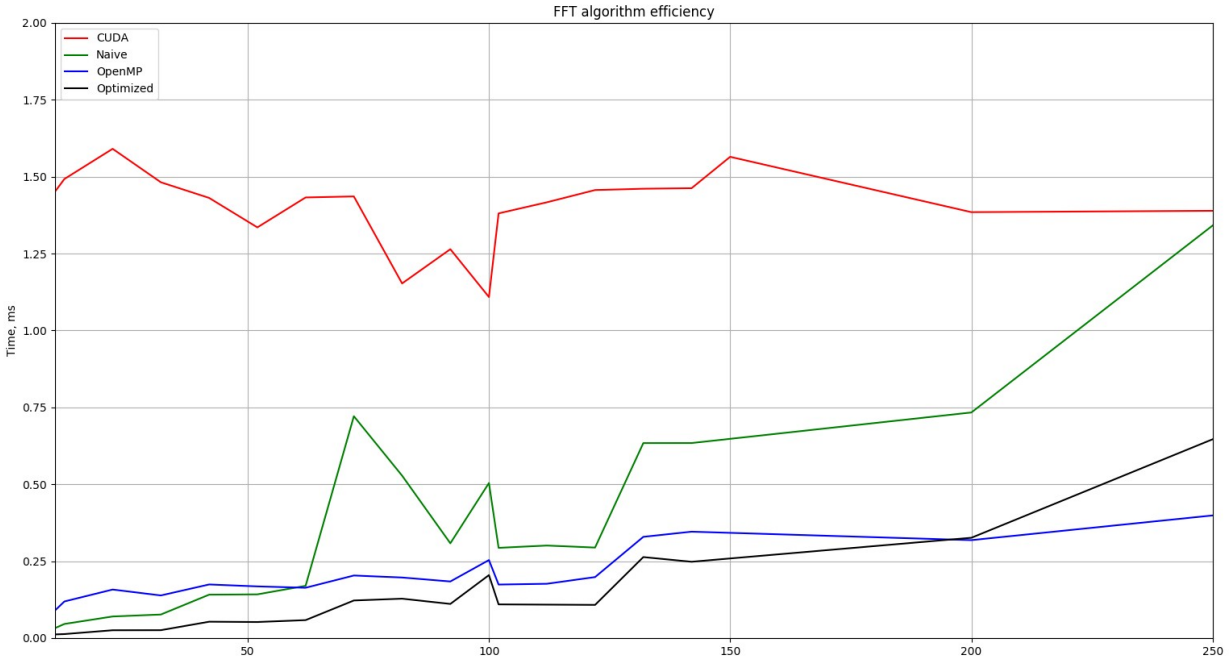

Рис 5. – График результатов тестирования на очень маленьких объемах данных.

Полученные результаты свидетельствуют о том, что для вычислений при очень маленьком объеме данных выгоднее использовать более простые реализации – самым эффективным оказался алгоритм вычисления БПФ без рекурсии. Наивный алгоритм, очевидно, проигрывает из-за наличия рекурсивных вызовов. Алгоритм, использующий CUDA, проигрывает в производительности ввиду больших временных затрат на Host/Device transfers, необходимых для вычисления на GPU. Алгоритм, использующий OpenMP, неэффективен на маленьких объемах данных ввиду больших, чем на вычисления, затрат на синхронизацию потоков.

3.2. Тестирование на средних объемах данных.

Под средними объемами данных имеются в виду данные, помещающиеся в GPU компьютера. GPU ноутбука, на котором проводилось тестирование алгоритмов, имеет объем 256 Мб, что соответствует 2 целочисленным массивам, содержащим 67108864/2 элементов.

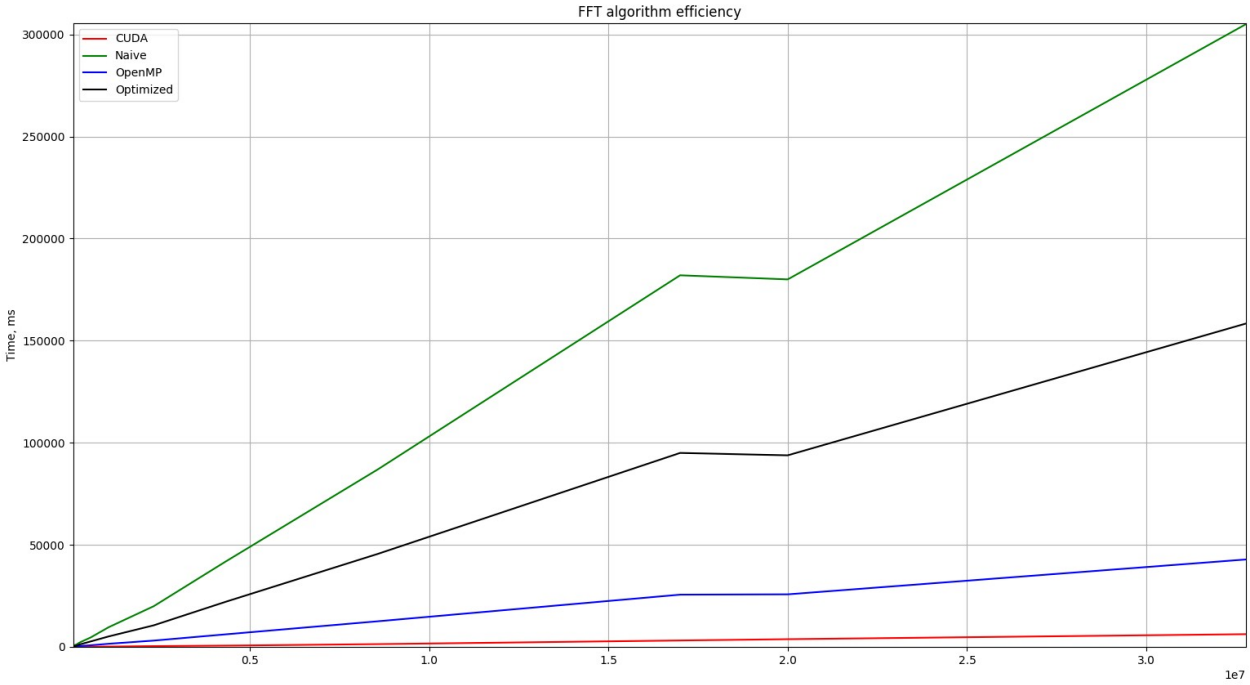

Рис 6. – График результатов тестирования на средних объемах данных.

В таблице на рисунке 4 также приводятся среднее время вычисления алгоритма при входных данных, помещающихся в GPU (первая строка), и отношение времени выполнения алгоритма к времени вычислений реализации с использованием CUDA.

Naive	Optimized	OpenMP	CUDA
43209.23714102564	22709.138871794872	6371.791179487179	842.5942625
51.281190798549545	26.951452059994143	7.562110808328912	1

Рис. 7 – Таблица сравнения результатов тестирования на средних объемах данных.

Проанализировав полученные результаты и сравнив их с результатами тестирования на маленьких объемах данных, очевидно, что вычисления на GPU стали еще более эффективными с увеличением объема входных данных.

3.3. Тестирование на больших объемах данных.

Под большими объемами данных имеются в виду данные, уже не помещающиеся в GPU компьютера, что означает объем входных данных, соответствующий 2 целочисленным массивам, содержащим более чем 67108864/2 элементов.

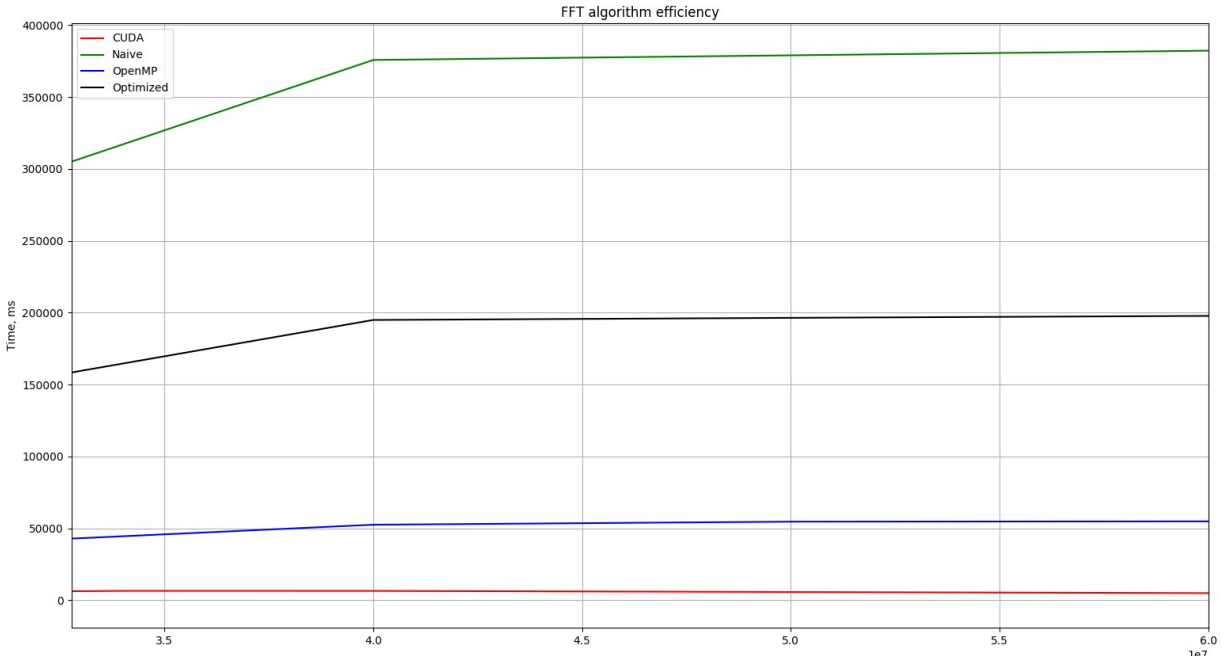

Рис 8. – График результатов тестирования на больших объемах данных.

В таблице на рисунке 4 также приводятся среднее время вычисления алгоритма при входных данных, не помещающихся в GPU (первая строка), и отношение времени выполнения алгоритма к времени вычислений реализации с использованием CUDA.

Naive	Optimized	OpenMP	CUDA
378960.3333333333	196327.33333333334	53971.56666666667	5005.3966666666665
75.71034996227405	39.22313183304154	10.782675232532355	1

Рис. 9 – Таблица сравнения результатов тестирования на больших объемах данных.

Между реализациями оптимизированной и неоптимизированной версий БПФ прослеживается линейная зависимость. Оптимизированная реализация быстрее неоптимизированной примерно в 2 раза. Реализация распараллеливания на CPU примерно в 3.6 раза быстрее оптимизированной на больших объемах данных, и это соотношение несколько (не слишком значительно) увеличивается с увеличением объема данных.

Вновь очевидно, что реализация же на CUDA становится заметно эффективнее с увеличением объема данных по сравнению с прочими реализациями, что подтверждают результаты таблиц на рисунках 3, 7 и 9.

4. Выводы.

Выполнение расчетов на GPU показывает отличные результаты в алгоритмах, использующих параллельную обработку данных, когда одну и ту же последовательность математических операций применяют к большому объему данных. Так как пересылка данных между устройствами (Host/Device transfers) требует значительных временных затрат и представляет собой наиболее медленное действие при рассмотрении вычислений на GPU, на маленьких объемах данных использовать алгоритм невыгодно. При вычислениях же на больших размерах данных вычисления на GPU не только опережают все прочие, но и продолжают наращивать преимущество с ростом размерности.

Список использованной литературы

1. Fast Fourier Transforms (FFTs) and Graphical Processing Units (GPUs) [Электронный ресурс]. - Электронные данные. - Режим доступа: (http://users.umiacs.umd.edu/~ramani/cmsc828e_gpusci/DeSpain_FFT_Presentation.pdf)
2. CUDA: CUFFT Library [Электронный ресурс]. - Электронные данные. - Режим доступа: (http://developer.download.nvidia.com/compute/cuda/1.0/CUFFT_Library_1.0.pdf)
3. High Throughput Long Integer Multiplication using Fast Fourier Transform on Parallel Workstation [Электронный ресурс]. - Электронные данные. - Режим доступа: (https://www.researchgate.net/publication/272122279_High_Throughput_Long_Integer_Multiplication_using_Fast_Fourier_Transform_on_Parallel_Workstation)
4. Алгоритмы[Электронный ресурс] - e-maxx.ru.