УДК 658.512

Ю.Ю. Ланкевич

АЛГОРИТМЫ СБОРКИ ОБЩЕГО ИЗОБРАЖЕНИЯ ТОПОЛОГИИ СБИС

Рассматривается задача сборки общего изображения слоя топологии сверхбольших интегральных микросхем из кадров, полученных путем фотографирования микроскопом технологического слоя микросхемы. Предлагается использование графических ускорителей и возможностей технологии CUDA для обработки большого объема информации. Программно реализуются алгоритмы сборки общего видеоизображения.

Введение

В настоящее время существует необходимость в анализе изображений топологии полупроводниковых сверхбольших интегральных микросхем (СБИС), спроектированных по субмикронным нормам.

Одной из задач, которые необходимо решить, чтобы выполнить анализ микросхемы, является сборка общего изображения топологии СБИС из отдельных кадров. Множество кадров для анализа, которые соответствуют фрагментам слоев микросхемы, получают путем фотографирования кристалла слой за слоем с помощью электронного микроскопа.

Для выполнения анализа видеоизображений топологии СБИС требуются программные средства поддержки указанного процесса. Качество анализа во многом будет определяться набором используемых инструментов и их возможностями [1]. Так как полученные с помощью микроскопа изображения представляют собой небольшие фрагменты топологии схемы, то необходимы программные средства построения общего изображения слоя топологии СБИС. Такие программные средства включают в себя программу поиска оптимального совмещения соседних кадров (по горизонтали и вертикали) и программу объединения кадров в общее изображение слоя топологии СБИС, реализующую алгоритм оценки качества полученного изображения и алгоритм исправления искажений, связанных с ошибкой совмещения.

Обработкой изображений в настоящее время занимается достаточно много специалистов в различных областях: в области картографии поверхности земли и морского дна [2], в медицине [3] и других, в том числе в области обработки изображений СБИС [4, 5].

Обработка изображений СБИС включает в себя получение общего видеоизображения топологии СБИС, выделение контуров объектов [6-10], локализацию и классификацию объектов [7, 11–14].

В настоящей работе рассматривается задача получения общего видеоизображения топологии СБИС, основное отличие предлагаемого подхода заключается в использовании параллельных алгоритмов и их реализации на графических ускорителях, что сильно уменьшает время обработки большого числа изображений.

1. Постановка задачи

Множество кадров изображения топологии СБИС необходимо объединить в одно общее изображение с минимальной ошибкой совмещения. Дополнительным требованием является разработка параллельных алгоритмов решения задачи с использованием технологии параллельных вычислений CUDA (Compute Unified Device Architecture).

Решение задачи предлагается осуществлять в три этапа:

1) построение матриц ошибок совмещений;

2) анализ матриц ошибок совмещений;

3) коррекция общего видеоизображения по результатам анализа и вычисление относительных координат для каждого кадра.

В отличие от работы [4] на этапе 1 предлагается полный перебор возможных совмещений соседних кадров при поиске их наилучшего совмещения, что позволяет получать более качественное решение задачи в целом.

2. Основы создания программ на CUDA

Рассмотрим базовые архитектурные особенности видеочипов фирмы NVIDIA с технологией CUDA [15, 16], на которых реализуются разрабатываемые алгоритмы работы с изображениями. Графический процессор (Graphics Processing Unit, GPU) включает несколько кластеров текстурных блоков (Texture Processing Cluster). Каждый кластер состоит из укрупненного блока текстурных выборок и двух-трех потоковых мультипроцессоров, каждый из которых представляет собой восемь вычислительных устройств и два суперфункциональных блока. Все инструкции выполняются по принципу SIMD (Single Instruction, Multiple Data – одна инструкция, много данных), когда одна инструкция применяется к большому количеству данных. Этот способ выполнения назвали SIMT (Single Instruction, Multiple Threads – одна инструкция, много потоков).

Каждый из мультипроцессоров имеет определенные ресурсы. Так, есть специальная разделяемая память объемом 16 Кбайт на мультипроцессор, но это не кеш, так как программист может использовать ее для любых нужд. Разделяемая память позволяет обмениваться информацией между потоками одного блока. Важно, что все потоки одного блока всегда выполняются одним и тем же мультипроцессором. Потоки из разных блоков обмениваться данными не могут, и необходимо помнить об этом ограничении. Мультипроцессоры могут обращаться и к видеопамяти, но с большими задержками и худшей пропускной способностью. Для ускорения доступа и снижения частоты обращения к видеопамяти у мультипроцессоров есть по 8 Кбайт кеша на константы и текстурные данные.

Мультипроцессор использует 8192–16 384 (для ускорителей G8x/G9x и GT2xx соответственно) регистров, общих для всех потоков всех блоков, выполняемых на нем. Максимальное число блоков на один мультипроцессор для G8x/G9x равно восьми, а число групп по 32 потока – 24 (768 потоков на один мультипроцессор). Всего видеокарты серий GeForce 8 и 9 могут обрабатывать до 12 288 потоков единовременно. GeForce GTX 280 на основе GT200 предлагает до 1024 потоков на мультипроцессор, в нем есть 10 кластеров по три мультипроцессора, обрабатывающих до 30 720 потоков. Знание этих ограничений позволяет оптимизировать алгоритмы под доступные ресурсы.

Первым шагом при переносе существующего приложения на CUDA является его профилирование и определение участков кода, являющихся узким местом, тормозящим работу. Если среди таких участков есть подходящие для быстрого параллельного исполнения, эти функции переносятся на C-расширения CUDA для выполнения на GPU. Программа компилируется при помощи поставляемого фирмой NVIDIA компилятора, который генерирует код и для CPU, и для GPU. При исполнении программы центральный процессор выполняет свои порции кода, а GPU выполняет CUDA-код с наиболее тяжелыми параллельными вычислениями. Эта часть программы, предназначенная для выполнения на GPU, называется ядром (kernel). В ядре определяются операции, которые будут исполнены над данными.

Видеочип получает ядро и создает копии для каждого элемента данных. Эти копии называются потоками (thread). Поток содержит счетчик, регистры и состояние. Для больших объемов данных, таких как обработка изображений, запускаются миллионы потоков. Потоки выполняются группами по 32 штуки, называемыми варпами. Варпам назначается исполнение на определенных потоковых мультипроцессорах. Каждый мультипроцессор состоит из восьми ядер – потоковых процессоров, которые выполняют одну инструкцию за один такт.

Мультипроцессор не является традиционным многоядерным процессором, он отлично приспособлен для многопоточности, поддерживая до 32 варпов единовременно. Каждый такт аппаратное обеспечение выбирает, какой из варпов исполнять, и переключается от одного к другому без потерь в тактах.

GPU (Graphics Processing Unit – графический процессор) является вычислительным устройством, сопроцессором (device) для центрального процессора (host), обладающим собствен-

ной памятью и обрабатывающим параллельно большое количество потоков. Ядром называется функция для GPU, исполняемая потоками (аналогия из 3D-графики – шейдер). Модель программирования в CUDA предполагает группирование потоков. Потоки объединяются в блоки потоков (thread block) – одномерные или двухмерные сетки потоков, взаимодействующих между собой при помощи разделяемой памяти и точек синхронизации. Программа (ядро, kernel) исполняется над сеткой (grid) блоков потоков (рис. 1). Одновременно исполняется одна сетка. Каждый блок может быть одно-, двух- или трехмерным по форме и может состоять из 512 потоков для существующего в настоящее время аппаратного обеспечения.

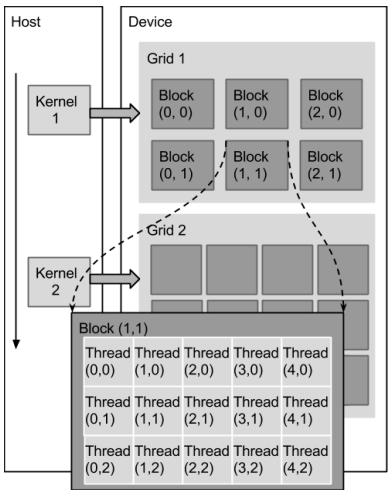


Рис. 1. Структура потоков в CUDA

Блоки потоков выполняются в виде варпов. В связи с тем что не всегда удобно работать с 32 потоками, CUDA позволяет работать и с блоками, содержащими от 64 до 512 потоков.

Группировка блоков в сетки позволяет уйти от ограничений и применить ядро к большему числу потоков за один вызов. Это помогает и при масштабировании. Если у GPU недостаточно ресурсов, он будет выполнять блоки последовательно. В противном случае блоки могут выполняться параллельно, что важно для оптимального распределения работы на видеочипах разного уровня, начиная от мобильных и интегрированных.

Модель памяти в CUDA имеет возможность побайтной адресации. Доступно довольно большое количество регистров на каждый потоковый процессор – до 1024 штук. Доступ к ним очень быстрый, хранить в них можно 32-битные целые числа или числа с плавающей точкой. Каждый поток имеет доступ к различным типам памяти (рис. 2).

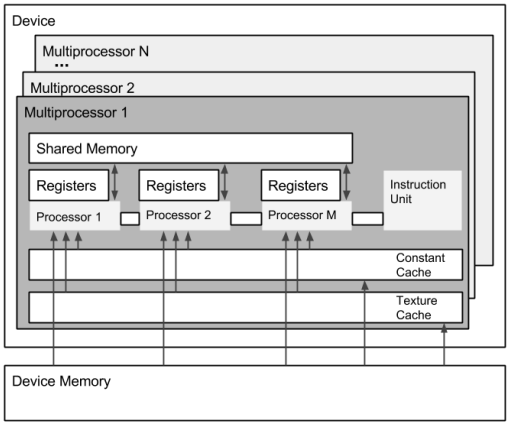


Рис. 2. Структура памяти в CUDA

Глобальная память (Global memory) – самый большой объем памяти, доступный для всех мультипроцессоров на GPU. Ее размер составляет от 256 Мбайт до 8 Гбайт. Обладает высокой

пропускной способностью (более 100 Гбайт/с для современных решений NVIDIA) и имеет очень большие задержки в несколько сот тактов.

Локальная память (Local memory) – небольшой объем памяти, к которому имеет доступ только один потоковый процессор. Она относительно медленная – такая же, как и глобальная.

Разделяемая память (Shared memory) – 16-килобайтный блок памяти с общим доступом для всех потоковых процессоров в мультипроцессоре. Эта память имеет такую же, как и регистры, скорость доступа. Она обеспечивает взаимодействие потоков, управляется разработчиком напрямую и имеет низкие задержки. Преимущества разделяемой памяти: использование в виде управляемого программистом кеша первого уровня, снижение задержек при доступе исполнительных блоков к данным, сокращение количества обращений к глобальной памяти.

Память констант (Constant memory) – область памяти объемом 64 Кбайт, доступная только для чтения всеми мультипроцессорами. Она кешируется по 8 Кбайт на каждый мультипроцессор. Имеет задержку в несколько сот тактов при отсутствии нужных данных в кеше.

Текстурная память (Texture memory) – блок памяти, доступный для чтения всеми мультипроцессорами. Выборка данных осуществляется при помощи текстурных блоков видеочипа, поэтому предоставляются возможности линейной интерполяции данных без дополнительных затрат. Кешируется по 8 Кбайт на каждый мультипроцессор. Медленная, как и глобальная: имеет сотни тактов задержки при отсутствии данных в кеше.

Естественно, что глобальная, локальная, текстурная и память констант – это физически одна и та же память, известная как локальная видеопамять видеокарты. Их отличия состоят в различных алгоритмах кеширования и моделях доступа. Центральный процессор может обновлять и запрашивать только внешнюю память: глобальную, константную и текстурную (рис. 3).

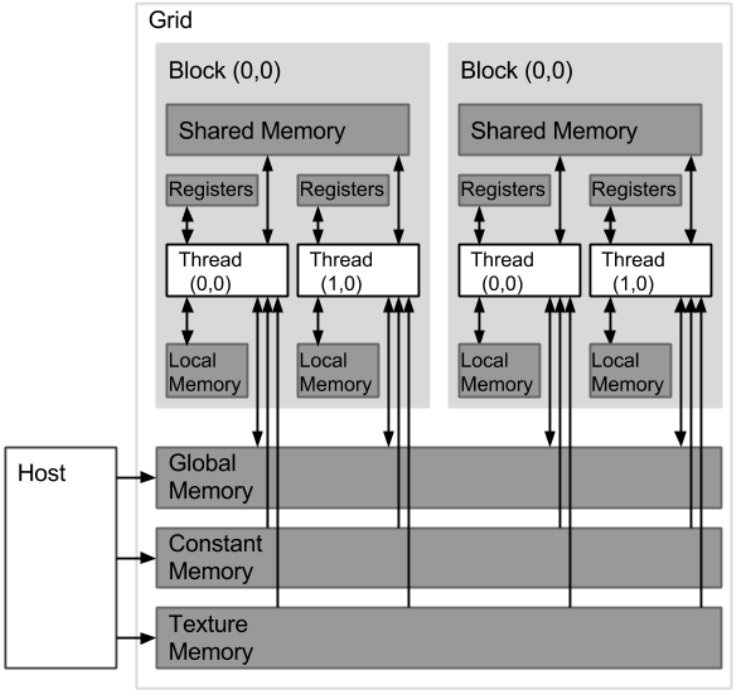


Рис. 3. Структура потоков в CUDA

При работе с графическими ускорителями нужно помнить о разных типах памяти, в частности о том, что локальная память и глобальная не кешируются и задержки при доступе к ним гораздо выше, чем у регистровой памяти, так как она физически находится в отдельных микросхемах. Таким образом, технология CUDA предполагает специальный подход к разработке – не совсем такой, как принят в программах для CPU.

3. Алгоритм построения матриц ошибок совмещений соседних кадров (этап 1)

Прежде чем сформулировать задачу, решаемую на этапе 1, определим некоторые понятия: перекрытие – общая область соседних кадров, совмещение – позиция одного кадра относительно другого.

Входными данными для алгоритма построения матриц ошибок совмещений соседних кадров являются:

– множество кадров (все кадры имеют одинаковые размеры) изображения топологии СБИС в формате *.dds или *.bmp;

– упорядоченный список имен файлов кадров, начиная с первого кадра первой строки и заканчивая последним кадром последней строки;

– количество кадров в строке и в столбце;

– среднее перекрытие для правого и для нижнего совмещений;

– максимальное отклонение от среднего перекрытия.

Алгоритм основывается на том, что оценить ошибку каждого совмещения для перекрытия двух кадров можно, вычислив сумму разниц значений соответствующих пикселов двух кадров по компонентам цвета. Кадры представляются в рамках цветовой модели RGB (Red, Green, Blue), т. е. каждый пиксел имеет три компонента цвета. Каждый элемент результирующей матрицы ошибок представляет собой суммарную ошибку для одного совмещения двух соседних кадров.

В качестве примера выберем один из возможных форматов обрабатываемых изображений BMP (Bitmap Picture). В формате BMP левый нижний угол изображения находится в начале массива данных (рис. 4). Далее все выражения выводятся для этого случая.

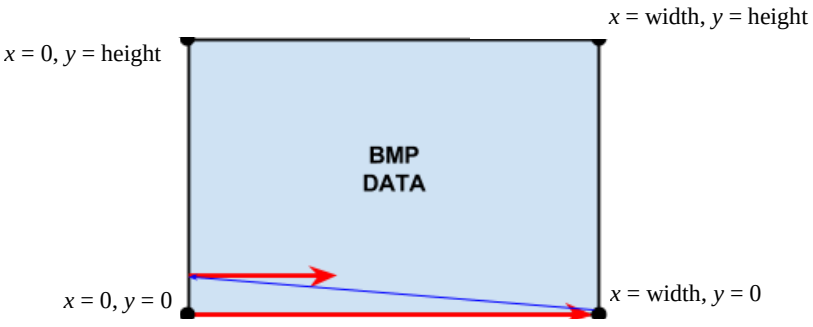


Рис. 4. Координаты пикселов в BMP-изображении

Рассмотрим пример горизонтального совмещения соседних кадров. На рис. 5 xmargin представляет собой наибольшее перекрытие кадров по горизонтали. Оно равно сумме среднего перекрытия по горизонтали и максимально допустимого отклонения перекрытия от среднего (offset).

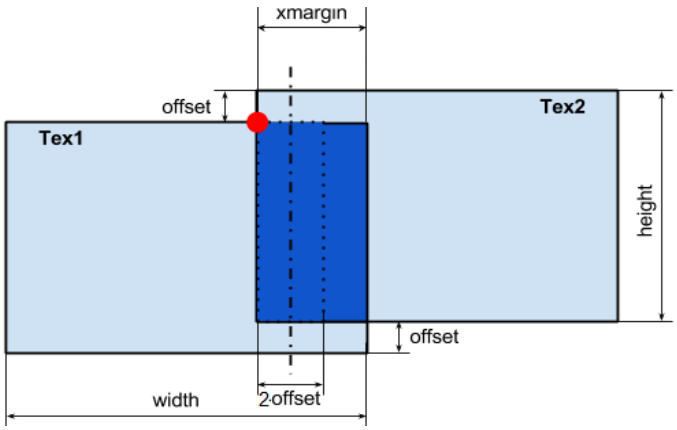


Рис. 5. Горизонтальное совмещение соседних кадров

Далее будем использовать следующую сокращенную запись условных выражений:

$$Y = A \ ? \ B : C,$$

интерпретируемых так: если условие A выполняется, то $Y = B$, если же условие A не выполняется, то $Y = C$.

При горизонтальном сравнении координаты левого верхнего угла для левого кадра Tex1 при заданном смещении Δx и Δy вычисляются по формулам

$$X_{\text{Tex1Pixel}} = \text{width} - 1 - \text{xmargin} + \Delta x;$$

$$Y_{\text{Tex1Pixel}} = (\Delta y < \text{offset}) \ ? \ \text{height} - 1 : (\text{height} - 1) - (\Delta y - \text{offset}),$$

где Δx и Δy – смещение перекрытия по горизонтали и вертикали, которое изменяется в дискретном диапазоне от 0 до 2*offset – 1, единица дискретизации равна одному пикселу; width – ширина изображения кадра; height – высота изображения кадра; offset – максимально допустимое отклонение перекрытия от среднего.

Координаты левого верхнего угла для правого кадра Tex2 при заданном смещении Δx и Δy вычисляются по формулам

$$X_{\text{Tex2Pixel}} = 0;$$

$$Y_{\text{Tex2Pixel}} = (\Delta y < \text{offset}) \ ? \ (\text{height} - 1) - (\text{offset} - \Delta y) : \text{height} - 1.$$

Ширина xwidth и высота yheight окна перекрытия кадров для заданных (Δx, Δy) определяются выражениями

$$\text{xwidth} = \text{xmargin} - \Delta x;$$

$$\text{yheight} = (\Delta y < \text{offset}) \ ? \ \text{height} - \text{offset} + \Delta y : \text{height} - (\Delta y - \text{offset}).$$

Ошибка совмещения кадров для заданного смещения (Δx, Δy)

$$\text{Err}(\Delta x, \Delta y) = \frac{\sum\limits_{i=0;\, j=0}^{i=\text{xwidth}-1;\, j=\text{yheight}-1} (R_{\text{Tex1Pixel}} - R_{\text{Tex2Pixel}}) + (G_{\text{Tex1Pixel}} - G_{\text{Tex2Pixel}}) + (B_{\text{Tex1Pixel}} - B_{\text{Tex2Pixel}})}{\text{yheight} \cdot \text{xwidth}},$$

где $R_{\text{TexXPixel}}$, $G_{\text{TexXPixel}}$, $B_{\text{TexXPixel}}$ (X = 1, 2) – три компонента цвета пиксела в левом и правом изображении кадров для заданных смещений i, j в окне.

Задача совмещения, решаемая на этапе 1: необходимо рассчитать значения ошибок совмещения для всех возможных перекрытий (Δx, Δy) соседних кадров, т. е. всего 4*offset^2 (Δx и Δy могут принимать значения от 0 до 2*offset), после чего определить смещение (Δx, Δy), для которого $\text{Err}_{(\Delta x,\, \Delta y)}$ будет минимальным.

Для вертикального совмещения (рис. 6) координаты левого верхнего угла для верхнего изображения Tex1 при заданном смещении Δx и Δy вычисляются по формулам

$$X_{\text{Tex1Pixel}} = (\Delta x < \text{offset}) \ ? \ 0 : \Delta x - \text{offset};$$

$$Y_{\text{Tex1Pixel}} = \text{ymargin} - \Delta y,$$

где ymargin – наибольшее перекрытие кадров по вертикали.

Координаты левого верхнего угла для нижнего кадра Tex2 при заданном смещении Δx и Δy вычисляются по формулам

$$X_{\text{Tex2Pixel}} = (\Delta x < \text{offset}) \ ? \ \text{offset} - \Delta x : 0 \ ;$$

$$Y_{\text{Tex2Pixel}} = \text{height} - 1.$$

Ширина и высота окна перекрытия кадров определяются выражениями

$$xwidth = (\Delta x < offset)\ ?\ width - offset + \Delta x : width - (\Delta x - offset);$$

$$yheight = ymargin - \Delta y.$$

Расчет матрицы ошибок Err для вертикального совмещения выполняется с учетом приведенных выше выражений.

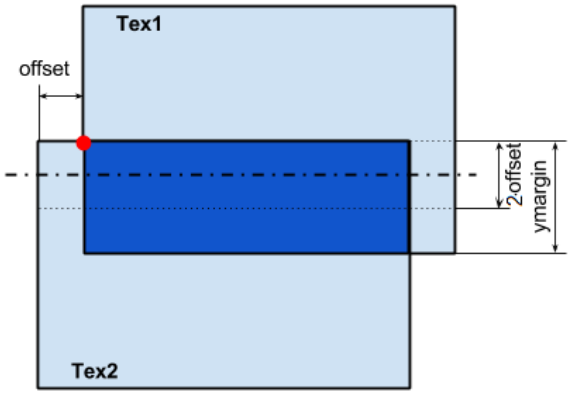


Рис. 6. Вертикальное совмещение соседних кадров

Отдельный поток выполнения программы называется thread. Потоки (не более 1024 штук) объединяются в блоки. Идентифицировать поток в блоке можно по трем координатам threadIdx.x, threadIdx.y, threadIdx.z, максимальные значения (blockDim.x, blockDim.y, blockDim.z) которых задаются пользователем перед запуском задачи на GPU. Блоки объединяются в сеть, в которой блок можно определить также по трем координатам blockIdx.x, blockIdx.y, blockIdx.z, максимальные значения которых задаются параметрами gridDim.x, gridDim.y, gridDim.z. На рис. 7 показано разделение блоков на потоки, используемое для реализации алгоритма нахождения лучшего совмещения.

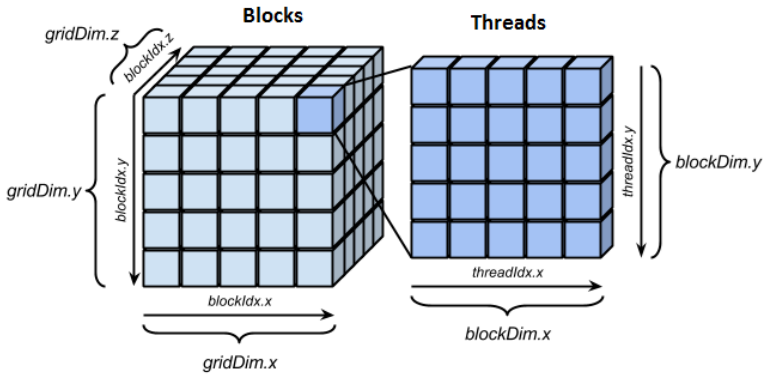


Рис. 7. Разделение параллельных потоков на блоки и потоки

По координатам blockIdx.x, blockIdx.y задаются различные смещения (Δx, Δy) по горизонтали и вертикали перекрытия соседних кадров, в этом случае blockDim.x = 2 × offset, blockDim.y = 2 × offset. Окно перекрытия соседних кадров (рис. 8) делится на квадратные области по 16×16 пикселов, которые представляют один блок по координате blockIdx.z. На рис. 8 показан пример для случая, когда $nx = 5$, $ny = 10$. Таким образом, для расчета координат пиксела в окне перекрытия кадров используются выражения

$$divx = (blockIdx.z \% nx) * blockDim.x + threadIdx.x;$$

$$divy = (blockIdx.z / nx) * blockDim.y + threadIdx.y.$$

Каждый поток рассчитывает значение ошибки (разницы между цветами пикселов) для одного отдельного пиксела, сохраняя это значение в разделяемой памяти, общей для одного блока. После того как все потоки в блоке рассчитают значения ошибки, она суммируется с помощью стандартного параллельного алгоритма суммирования путем редукции [15, с. 75]. Синхронизация потоков в блоке осуществляется с помощью функции __syncthreads(). Значение суммарной ошибки по блоку 16×16 пикселов записывается в глобальную память GPU. Результирующий массив ошибок снова суммируется с помощью редукции, и результат сохраняется в файл. Формат файла матриц ошибок представляет собой двухмерный массив, содержащий $4*offset^2$ элементов типа float.

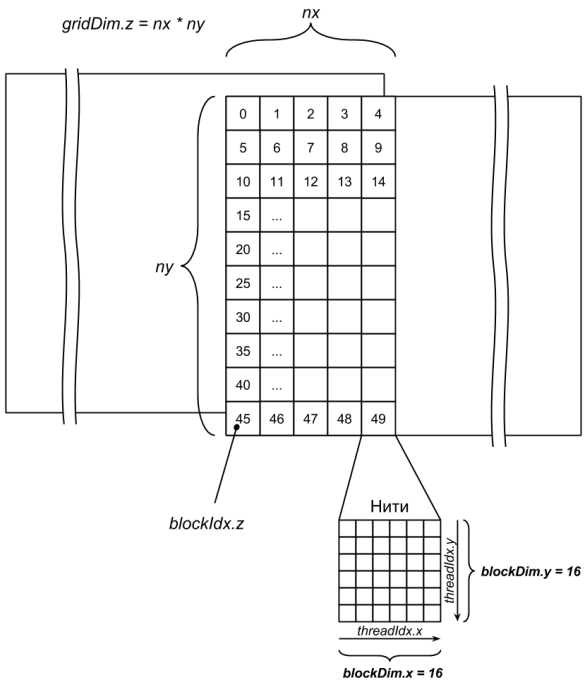


Рис. 8. Разбиение вычислений на блоки и потоки при горизонтальном совмещении соседних кадров

4. Алгоритм анализа матриц ошибок совмещений (этап 2)

Задача анализа матриц ошибок (задача этапа 2) состоит в том, чтобы по матрицам ошибок получить матрицы относительных координат соседних кадров и матрицы приоритетов, которые соответствуют полученным относительным координатам.

Алгоритм анализа матриц ошибок совмещений обрабатывает исходную информацию, заданную в виде матриц ошибок совмещений соседних кадров. Результатом выполнения алгоритма решения задачи анализа матриц ошибок совмещений являются матрицы относительных координат соседних кадров и матрицы приоритетов, которые соответствуют полученным относительным координатам. Понятие приоритета будет рассмотрено далее.

Алгоритм анализа ошибок совмещений основывается на следующих утверждениях:

1. Каждый кадр (если он не крайний справа или снизу) имеет соседний кадр справа и снизу.
2. Для каждого соседнего кадра существует матрица совмещений M, каждый элемент которой соответствует позиции совмещения этого кадра с соседним, а значение соответствует ошибке совмещения (сумме разниц значений соответствующих пикселов совмещаемых кадров).
3. Среднее перекрытие для всех соседних кадров зависит от шага движения микроскопа, величина шага является примерно одинаковой в пределах одного слоя.
4. Каждая матрица совмещений M нормализуется так, что все ее элементы принимают значения от 0 до 1.
5. Для всех матриц M существует отклонение от минимального значения, в пределах которого находится искомое значение относительных координат взаимосовмещения соседних кадров, которым соответствует эта матрица.

6. Чем больше разброс значений, которые находятся в пределах заданного отклонения от минимального значения, тем менее достоверной является эта матрица и тем больше вероятность того, что относительная позиция соседнего кадра, соответствующая элементу из M с минимальным значением, не будет соответствовать искомой.

На рис. 9 в графическом виде представлены некоторые виды матриц совмещений в форме изображений в градациях серого. Черная точка соответствует нулю, а белая – единице. На рис. 9, *а* показаны наименее достоверные матрицы совмещений: на значения, полученные с помощью этих матриц, полагаться нельзя, на рис. 9, *б* – наиболее достоверные матрицы.

а) *б)*

Рис. 9. Виды матриц совмещений: *а)* менее достоверные матрицы; *б)* более достоверные матрицы

Минимум находится в наиболее темной части каждого изображения, соответствующего матрице совмещений M. Поэтому чем больше эта область, тем менее достоверной будет относительная координата соседнего кадра, которая соответствует найденному минимуму. Если таких областей в одной матрице несколько и они разбросаны по всему изображению, найденная относительная координата соседнего кадра будет недостоверной.

Введем понятие *веса относительной координаты* – чем меньше вес, тем больше вероятность того, что значение относительной координаты является верным. Значение веса относительной координаты в дальнейшем будет выступать в роли приоритета в порядке пересчета полученных на первом этапе значений относительных координат всех кадров изображения.

Значение веса, или приоритета, вычисляется следующим образом:

1. Выбирается пороговое значение, ниже которого все значения из матрицы ошибок считаются равнозначными.

2. Находятся наименьшая и наибольшая позиции (по горизонтали и по вертикали отдельно) элементов матрицы ошибок из множества элементов, значения которых меньше порогового значения.

3. Весу присваивается разница наименьшей и наибольшей позиций.

Потом вычисляются значение относительной координаты, которое соответствует минимуму в заданных границах матрицы ошибок, и среднее значение относительных координат с наименьшим значением веса относительных координат для ряда кадров и производится пересчет относительных координат в соответствии с правилом трех сигм [17].

5. Алгоритм коррекции общего видеоизображения по результатам анализа (этап 3)

Основная задача коррекции общего видеоизображения (задача этапа 3): по результатам анализа требуется пересчитать значения относительных координат всех совмещенных кадров общего видеоизображения.

Прежде чем описывать алгоритм решения задачи коррекции общего видеоизображения, рассмотрим общий случай (рис. 10) расположения четырех соседних кадров видеоизображений.

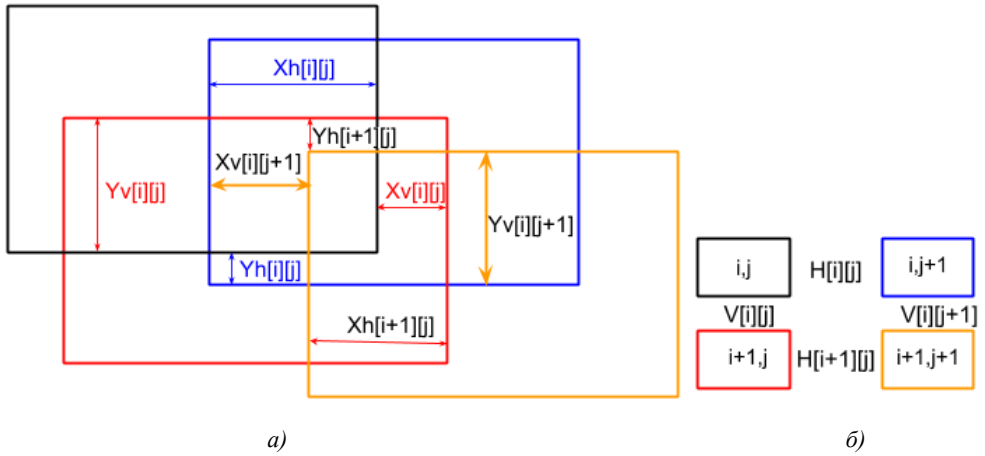
На этапе 2 были определены матрицы относительных координат (Xh и Xv – для правого и нижнего кадра соответственно по координате x; Yh и Yv – для правого и нижнего кадра соответственно по координате y) исходя из информации матриц ошибок совмещений.

Для четырех соседних кадров должны выполняться следующие условия:

$$Xv[i][j] + Xh[i][j] = Xv[i][j + 1] + Xh[i + 1][j]; \quad (1)$$

$$Yv[i][j] + Yh[i][j] = Yv[i][j + 1] + Yh[i + 1][j], \quad (2)$$

где i – позиция кадра в строке; j – позиция кадра в столбце; $Xv[i][j]$ – относительная координата по x для нижнего соседнего кадра; $Yv[i][j]$ – относительная координата по y для нижнего соседнего кадра; $Xh[i][j]$ – относительная координата по x для правого соседнего кадра; $Yh[i][j]$ – относительная координата по y для правого соседнего кадра.



а) *б)*

Рис. 10. Относительные координаты для четверки кадров (i – номер строки, j – номер столбца): *а)* кадры с перекрытием; *б)* положение кадров относительно друг друга

На этапе 2 были также определены веса относительных координат (matrix_pr_d[i][j][0] и matrix_pr_r[i][j][0] – для нижнего и правого кадра соответственно по координате x; matrix_pr_d[i][j][1] и matrix_pr_r[i][j][1] – для нижнего и правого кадра соответственно по координате y) и средние допустимые значения совмещений по x (M_value_x_r[i] и M_value_x_d[i] для правых и нижних кадров соответственно) и по y (M_value_y_r[i] и M_value_y_d[i] для правых и нижних кадров соответственно) для каждой строки кадров.

Алгоритм коррекции общего видеоизображения выполняется отдельно по координате x и отдельно по координате y.

Алгоритм для координаты x включает следующие шаги:

1. Определить четверку кадров с наименьшим суммарным весом относительных координат исходя из матриц matrix_pr_d и matrix_pr_r, где i:=0..$rows$ – 1, j:=0..$cols$ – 1. Четверка с минимальным (ненулевым) значением суммы

$$sum = \text{matrix\_pr\_r}[i][j][0] + \text{matrix\_pr\_d}[i][j][0] + \text{matrix\_pr\_r}[i + 1][j][0] + \text{matrix\_pr\_d}[i][j + 1][0]$$

будет соответствовать искомой четверке. Обозначим найденную позицию min_i_x и min_j_x. Если такой четверки не нашлось, то закончить.

2. Вычислить значение для каждой координаты из четырех, участвующих в пересчете, исходя из трех остальных, пользуясь (1), и вычислить разницу Err между полученным значением и средним по ряду кадров (M_value_x_r[i] или M_value_x_d[i]).

3. Сравнить значения Err, вычисленные в п. 2, со значением offset/4 (offset – максимальное отклонение от среднего перекрытия) по модулю и в зависимости от результата сравнения выполнить следующие действия:

– если нет ни одного значения из полученных разностей, которое меньше offset/4, то увеличить значения весов всех участвующих в пересчете относительных координат и перейти к п. 1;

– в противном случае для всех относительных координат, которые участвуют в пересчете, в порядке убывания соответствующих им значений |Err|, полученных в п. 2, выполнить следующее: если фиксация текущей координаты не мешает дальнейшему пересчету, то зафиксировать ее, присвоив соответствующему элементу из матриц приоритетов нуль.

4. Перейти к п. 1.

После присвоения какому-либо элементу из матриц приоритетов значения 0 производится пересчет всех возможных относительных координат с помощью выражения (1), исходя из того, что значение каждой относительной координаты, которой соответствует элемент из матрицы приоритетов, имеющий значение 0, понимается как константа. Соответствующему каждой пересчитанной относительной координате элементу матрицы приоритетов присваивается значение 0.

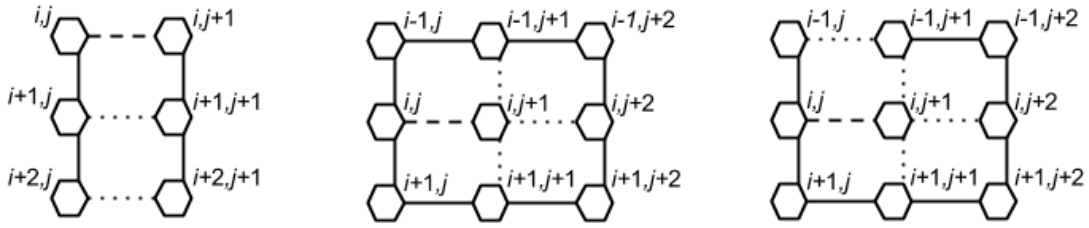
На шаге проверки, помешает ли фиксация координаты дальнейшему пересчету, сложность вызывает только проверка, будет ли иметь решение система линейных уравнений, которая определяется выражением (1) и нулевыми элементами из матриц приоритетов.

На рис. 11 шестигранник соответствует определенному кадру; позиция *i* (по строке), *j* (по столбцу) соответствует относительной координате, которая фиксируется в текущем цикле алгоритма. Сплошной линией на рисунке обозначены относительные координаты, которые были зафиксированы ранее (на предыдущих шагах алгоритма); пунктирной линией из точек – относительные координаты, которые пересчитываются в текущем состоянии; пунктирной линией из дефисов – относительная координата, которая фиксируется в текущем состоянии без пересчета.

Количество уравнений в системе можно подсчитать исходя из количества четверок соседних кадров. Количество переменных соответствует количеству пунктирных линий из точек.

В ситуации, представленной на рис. 11, *а*, решается система двух линейных уравнений с двумя переменными. В этом случае решение будет найдено всегда.

В ситуации, которая показана на рис. 11, *б*, решается система четырех линейных уравнений с тремя переменными: в этом случае решение может не существовать. Такую ситуацию необходимо избегать. Поэтому в алгоритме предусмотрена дополнительная проверка, основанная на анализе разрешимости системы линейных уравнений, приводимой ниже. В результате проверки меняется последовательность установки относительных координат.



а) *б)* *в)*

Рис. 11. Графическое представление системы линейных уравнений:
а) два уравнения с двумя переменными; *б)* четыре уравнения с тремя переменными;
в) четыре уравнения с четырьмя переменными

Фактически это означает, что рассматриваемая на текущем шаге алгоритма относительная координата не фиксируется и на последующих шагах вместо данной ситуации будет возникать ситуация, показанная на рис. 11, *в*. В этом случае решается система четырех линейных уравнений, где присутствуют четыре переменные, т. е. система уравнений будет иметь одно решение. Подчеркиванием обозначены те относительные координаты, которые пересчитываются, остальные считаются константами.

Система линейных уравнений на рис. 1, *а* имеет вид

$$Xv[i][j] + Xh[i][j] = Xv[i][j + 1] + \underline{Xh[i + 1][j]};$$

$$Xv[i + 1][j] + \underline{Xh[i + 1][j]} = Xv[i + 1][j + 1] + \underline{Xh[i + 2][j].}$$

Система линейных уравнений на рис. 11, *б* описывается как

$$Xv[i – 1][j] + Xh[i – 1][j] = \underline{Xv[i – 1][j + 1]} + Xh[i][j];$$

$$Xv[i][j] + Xh[i][j] = \underline{Xv[i][j+1]} + Xh[i+1][j];$$

$$Xv[i-1][j+1] + Xh[i-1][j+1] = Xv[i-1][j+2] + \underline{Xh[i][j+1]};$$

$$\underline{Xv[i][j+1]} + \underline{Xh[i][j+1]} = Xv[i][j+2] + Xh[i+1][j+1].$$

В случае, показанном на рис. 11, *в*, решается следующая система линейных уравнений:

$$Xv[i-1][j] + \underline{Xh[i-1][j]} = \underline{Xv[i-1][j+1]} + Xh[i][j];$$

$$Xv[i][j] + Xh[i][j] = \underline{Xv[i][j+1]} + Xh[i+1][j];$$

$$\underline{Xv[i-1][j+1]} + Xh[i-1][j+1] = Xv[i-1][j+2] + \underline{Xh[i][j+1]};$$

$$\underline{Xv[i][j+1]} + \underline{Xh[i][j+1]} = Xv[i][j+2] + Xh[i+1][j+1].$$

Алгоритм для координаты y имеет такие же шаги, что и для x, только вместо координаты x рассматривается координата y.

6. Экспериментальные данные

Все упомянутые в предыдущих разделах алгоритмы были программно реализованы. В течение наибольшего времени выполняется программа, реализующая алгоритм построения матриц ошибок, все остальные программы выполняются за сравнительно короткое время.

Приведем результаты выполнения программы построения матриц ошибок для различного числа кадров и различных параметров. В качестве ускорителя использовалась видеокарта GeForce Titan Black.

Пример 1. Количество кадров – 280, среднее перекрытие по x – 84, среднее перекрытие по y – 79, offset – 32.

Время выполнения для кадров в формате bmp – 1 мин 22 с, время выполнения для кадров в формате dds – 58 с.

Пример 2. Количество кадров – 6141, среднее перекрытие по x – 150, среднее перекрытие по y – 150, offset – 40.

Время выполнения для кадров в формате bmp – 66 мин 29 с, время выполнения для кадров в формате dds – 60 мин 45 с.

Заключение

В статье описаны алгоритмы для объединения кадров в общее изображение топологии СБИС. Отличительной особенностью предложенных алгоритмов является то, что они полностью ориентированы на возможности графических ускорителей и технологии CUDA для увеличения быстродействия. Использование параллельных алгоритмов и графических ускорителей позволило ускорить вычисления, что является первоочередным требованием для задач обработки большого объема информации, решение которых на обычных процессорах занимает слишком много времени. Разработанное на основе предложенных алгоритмов программное обеспечение является частью комплекса программ, необходимых для обеспечения технологического цикла производства СБИС как на этапе повторного проектирования, так и на этапе проверки полученных изделий. Экспериментальные данные свидетельствуют о высоком быстродействии разработанных программ.

Список литературы

1. НТЦ «Белмикросистемы» [Электронный ресурс]. – 2011. – Режим доступа : http://www.integral.by/ files/files/bms-2011.pdf. – Дата доступа : 01.03.2015.

2. Камаев, А.Н. Создание панорамных карт подводного дна на основе больших массивов изображений / А.Н. Камаев. – Владивосток, 2013. – С. 298–301.

3. Недзьведь, А.М. Анализ изображений для решения задач медицинской диагностики / А.М. Недзьведь, С.В. Абламейко. – Минск : ОИПИ НАН Беларуси, 2012. – 240 с.

4. Дудкин, А.А. Обработка изображений в проектировании и производстве интегральных схем / А.А. Дудкин, Р.Х. Садыхов. – Минск : ОИПИ НАН Беларуси, 2008. – 270 с.

5. Абламейко, С.В. Обработка изображений: технология, методы, применение / С.В. Абламейко, Д.М. Лагуновский. – Минск : ОИПИ НАН Беларуси, 1999. – 300 с.

6. Прэтт, У. Цифровая обработка изображений / У. Прэтт. – М. : Мир, 1982. – Кн. 2. – 480 с.

7. Введение в контурный анализ; приложения к обработке изображений и сигналов / Я.А. Фурман [и др.]. – М. : Физматлит, 2003. – 592 с.

8. Yuen, P.C. A contour detection method: Initialization and contour model / P.C. Yuen, G.C. Feng, J.P. Zhou // Pattern Recognition Letters. – 1999. – Vol. 20. – P. 141–148.

9. Koshchan, A. A Comparative Study on Color Edge Detection / A. Koshchan // Pattern Recognition Letters. – 2001. – Vol. 22, № 13. – P. 1419–1429.

10. Rosin, P.L. Edges: sliency measures and automatic tresholding / P.L. Rosin. – Mach. Vision, 1997. – P. 139–159.

11. Методы компьютерной обработки изображений / под. ред. В.А. Сойфера. – М. : Физматлит, 2003. – 784 с.

12. Kerfoot, I.B. Theoretical analysis of multispectral image segmentation criteria / I.B. Kerfoot, Y. Bresler // IEEE Trans. Image Processing. – 1999. – Vol. 8, № 6. – P. 768–820.

13. Coleman, G.B. Image Segmentation by Clustering / G.B. Coleman, H.C. Andrews // Proc. IEEE. – 1979. – Vol. 67. – P. 773–785.

14. Zhang, Z. A survey on evaluation methods for image segmentation / Z. Zhang // Pattern Recognition. –1996. – Vol. 29(8). – P. 1335–1346.

15. Сандерс, Д. Технология CUDA в примерах: введение в программирование графических процессоров / Д. Сандерс, Э. Кэндрот. – М. : ДМК Пресс, 2011. – 232 с.

16. Боресков, А.В. Основы работы с технологией CUDA / А.В. Боресков, А.А. Харламов. – М. : ДМК Пресс, 2010. – 232 с.

17. Боровиков, В. STATISTICA. Искусство анализа данных на компьютере: для профессионалов / В. Боровиков. – СПб. : Питер, 2003. – 688 с.

Поступила 14.05.2015

Объединенный институт проблем
информатики НАН Беларуси,
Минск, Сурганова, 6
e-mail: yury.lankevich@newman.bas-net.by

Y.Y. Lankevich

AN ALGORITHM FOR ASSEMBLING A COMMON IMAGE OF VLSI LAYOUT

We consider problem of assembling a common image of VLSI layout. Common image is composed of frames obtained by electron microscope photographing. Many frames require a lot of computation for positioning each frame inside the common image. Employing graphics processing units enables acceleration of computations. We realize algorithms and programs for assembling a common image of VLSI layout. Specificity of this work is to use abilities of CUDA to reduce computation time. Experimental results show efficiency of the proposed programs.