Санкт-Петербургский государственный университет

Кафедра системного программирования
Программная инженерия

Володин Вадим Евгеньевич

Реализация алгоритма Stereo Block Matching на архитектуре SIMD и VLIW

Курсовая работа

Научный руководитель:
ст. преп. Пименов А. А.

Санкт-Петербург
2019

Оглавление

Введение

Построение *карты диспаритетов* (disparity map) по стереопаре является важной задачей компьютерного зрения. Это - процесс преобразования двух изображений с восстановлением информации о расстоянии от точек сцены до картинной плоскости. *Карта диспаритетов* содержит данную информацию в качестве интенсивности пикселя (рис. 1). Большее значение интенсивности означает меньшее значение расстояния до картинной плоскости. *Карта диспаритетов* применяется в восстановлении 3D-модели объекта по нескольким его фотографиям, снятым с разных ракурсов; для выделения фона на изображениях; склейки панорам.

Основные методы решения данной задачи разделяются на глобальные и локальные [10]. Глобальные выигрывают в качестве у локальных, но проигрывают им в скорости. Некоторые алгоритмы компьютерного зрения могут быть реализованы эффективно на специализированном оборудовании - видеокартах или цифровых сигналных процессорах.

Одной из архитектур является линейка процессоров *EV6x* [3], произведенная компанией *Synopsys*. Ускорение происходит за счет совмещенной архитектуры *SIMD* [7] (Single Instruction Multiple Data) и *VLIW* [4] (Very Long Instruction Word). На данном процессоре реализован алгоритм построения *карты диспаритетов* Semi-Global Matching [6]. Он восстанавливает *карту диспаритетов*, обладающую хорошим качеством, но его скорость недостаточна. Необходим алгоритм с более высокой скоростью.

В рамках данной курсовой работы рассматривается Stereo Block Matching [2]. Целью данной работы является его реализация на архитектуре процессора *EV6x*.

Рис. 1: *Карта диспаритетов*, соответствующая стереопаре

1. Постановка задачи

Целью данной работы является реализация алгоритма Stereo Block Matching для архитектуры процессора *EV6x*.

Для достижения цели были поставлены следующие задачи:

- Изучение:
  - Алгоритмов стереозрения;
  - Алгоритма Stereo Block Matching;
  - Архитектуры процессора *EV6x*;
- Определение наилучшей асимптотики алгоритма;
- Построение алгоритма для данной архитектуры;
- Улучшение алгоритма;
- Реализация алгоритма.

2. Обзор

2.1. Описание алгоритма

Входом являются два изображения одной сцены с двух камер с разных точек обзора в конкретный момент времени.

Для корректной работы алгоритма изображения выравнивают так, чтобы для каждой точки соответствующая ей парная точка находилась в той-же строке на изображении со второй камеры. Данный процесс выравнивания изображений называют ректификацией.

Для каждого пикселя левого изображения выполняется поиск соответствующего пикселя на правом изображении. Если пиксель левого изображения имеет координаты (x_0, y_0), то соответствующий ему пиксель правого изображения будет иметь координаты $(x_0 - d, y_0)$, где d — величина, называемая смещением (*disparity*). Для каждого d от нуля до максимального значения смещения выделяются два квадратных блока одинакового размера (рис. 2), центрами которых являются соответствующие пиксели. Далее для блоков вычисляется метрика SAD (Sum of absolute differences).

$$sad_{y,x,d} = \sum_{i=-s}^{s} \sum_{j=-s}^{s} |L_{y+i,\, x+j} - R_{y+i,\, x+j+d}|$$

$$disparity_{y,x} = argmin_d \; sad_{y,x,d},$$

где L - левое изображение, R - правое.

Блок на правом изображении с наименьшим значением метрики определяет оптимальную пару.

Значение расстояния от точки сцены до картинной плоскости обратно пропорционально величине смещения:

$$\frac{T - d}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{f \cdot T}{d}$$

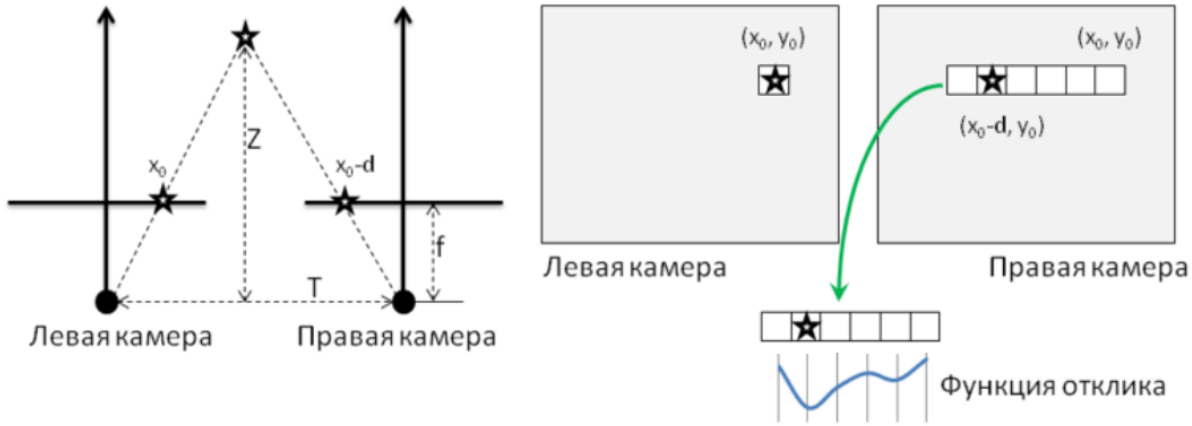

Рис. 2: Поиск смещения [11]

2.2. Архитектура процессора

Процессор (рис. 3), для которого необходимо реализовать данный алгоритм, имеет архитектуру *SIMD* с 512-битовыми регистрами. Поддерживаются типы int16, short32, char64, над которыми осуществляются операции с 16-ю числами типа int, 32-мя числами типа short и 64-мя числами типа char соответственно. Архитектура поддерживает *VLIW* инструкции, существует три слота для векторных операций, и один для скалярных. Таким образом, одновременно могут выполняться четыре инструкции. При этом некоторые типы инструкций, например, *vload* могут выполняться лишь на ограниченном наборе векторных слотов.

При разработке алгоритма следует учитывать количество векторной памяти, равное 128 КБ. Необходимо обрабатывать изображения по частям, не храня в памяти все данные. Например, размер *Full HD* изображения - 2 МБ, оно не поместится в векторную память.

На процессоре имеются 32 векторных регистра, в каждом из которых может хранится значение int16, short32 или char64. Некоторые регистры могут использоваться как аккумуляторные. В таком случае к ним добавляется некоторое количество дополнительных бит. К примеру, тип acc24x32 имеет дополнительные 8 бит у каждого из 32-х элементов вектора. Также существуют функции, работающие с предикатами. Они позволяют производить операции над заданными элементами вектора.

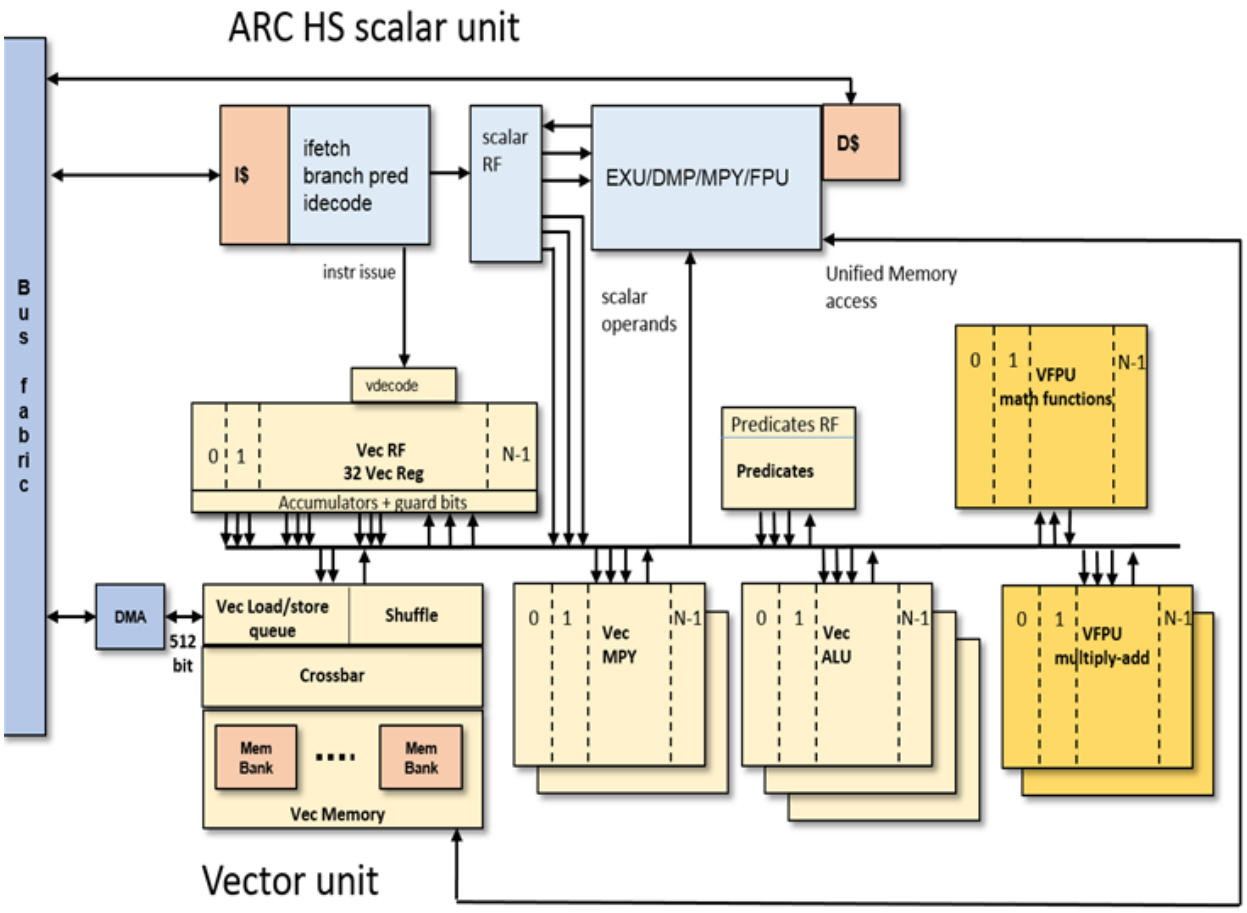

Рис. 3: Архитектура процессора *EV6x*

2.3. Обзор существующих решений

Алгоритм полного перебора имеет асимптотику:

$$O(w * h * b^2 * d),$$

где w и h - ширина и высота изображения, b - размер окна, для которого считается метрика SAD, d - максимальное смещение.

Метрику можно считать за константное время с предварительной обработкой [8]. Зная для данного смещения для каждого пикселя суммарное значение SAD на прямоугольнике, одним углом которого является пиксель с координатами $(0, 0)$, а вторым - данный пиксель, сумму на конкретном прямоугольнике можно вычислить методом скользящего окна. В таком случае асимптотика алгоритма составляет:

$$O(w * h * d)$$

Также существуют решения, использующие *SIMD*, такие как [5] и [1]. В [1] происходит одновременное вычисление нескольких значений *sad*.

3. Реализация

3.1. Инструменты

Для написания алгоритмов для процессора используется язык Си, так как программы на этом языке являются достаточно эффективными.

Также используется язык *OpenCL*, который является расширением языка Си. *OpenCL* - стандарт, разработанный группой *Khronos*. *OpenCL* реализуется производителями оборудования и позволяет разрабатывать приложения независимо от платформы. Таким образом избегается необходимость разработки приложений для каждой целевой платформы. Целью данной работы является написание *OpenCL* kernel-а (функция в OpenCL, входящая в интерфейс). В действительности, используется вариация *OpenCL* - *Metaware OpenCL*. Для тестирования используется тестовый набор данных Tsukuba [9] *Middlebury* колледжа и компании *Microsoft*. Этот набор данных является мировым стандартом для задач стереозрения. Для тестирования было реализовано два дополнительных решения. Первое - метод полного перебора, которое использовалось в качестве референсного. Это решение было затруднительно использовать для отладки основного алгоритма в силу его высокой вычислительной сложности, поэтому было реализовано второе решение - скалярная версия с улучшенной асимптотикой, результат которой сравнивался с результатом первого решения. Второе решение оказалось достаточно быстрым для отладки.

3.2. Скорость алгоритма и использование памяти

Для того, чтобы показать идею алгоритма и оценить используемую векторную память, введем величину $column\_sad$.

$$column\_sad_{y,x,d} = \sum_{i=-s}^{s} |L_{y+i,\,x} - R_{y+i,\,x+d}|$$

Покажем, как ее посчитать, зная $column\_sad_{y-1,x,d}$

$$column\_sad_{y,x,d} = \sum_{i=-s}^{s} |L_{y+i,\,x} - R_{y+i,\,x+d}| =$$

$$= |L_{y+s,\,x} - R_{y+s,\,x+d}| + \sum_{i=-s}^{s-1} |L_{y+i,\,x} - R_{y+i,\,x+d}| =$$

$$= |L_{y+s,\,x} - R_{y+s,\,x+d}| + \sum_{i=-s-1}^{s-1} |L_{y+i,\,x} - R_{y+i,\,x+d}| - |L_{y-s-1,\,x} - R_{y-s-1,\,x+d}| =$$

$$= |L_{y+s,\,x} - R_{y+s,\,x+d}| + \sum_{j=-s}^{s} |L_{y-1+j,\,x} - R_{y-1+j,\,x+d}| - |L_{y-s-1,\,x} - R_{y-s-1,\,x+d}| =$$

$$= |L_{y+s,\,x} - R_{y+s,\,x+d}| + column\_sad_{y-1,x,d} - |L_{y-s-1,\,x} - R_{y-s-1,\,x+d}|$$

Таким образом, *column_sad* для текущей строки зависит лишь от *column_sad* для предыдущей строки и пикселей изображения в пределах размера окна. При этом новое значение считается за константное время.

Покажем, как с помощью *column_sad* посчитать *sad*.

$$sad_{y,x,d} = \sum_{i=-s}^{s} column\_sad_{y,x+i,d} = column\_sad_{y,x+s,d} + \sum_{i=-s}^{s-1} column\_sad_{y,x+i,d} =$$

$$= column\_sad_{y,x+s,d} + \sum_{i=-s-1}^{s-1} column\_sad_{y,x+i,d} - column\_sad_{y,x-s-1,d} =$$

$$= column\_sad_{y,x+s,d} + \sum_{j=-s}^{s} column\_sad_{y,x-1+j,d} - column\_sad_{y,x-s-1,d} =$$

$$= column\_sad_{y,x+s,d} + sad_{y,x-1,d} - column\_sad_{y,x-s-1,d} =$$

Таким образом, *sad* также считается за константное время. При этом используя лишь *column_sad* с единственным индексом y.

Пересчитывая для каждого значения y сначала значения *column_sad*, а затем, зная эти значения, вычисляя *sad* для данного пикселя и данного смещения, получаем асимптотику:

$$O(w * h * d)$$

При этом используя дополнительное количество памяти для $column\_sad$, равное $w * d$.

Повышению эффективности способствуют одновременный подсчет sad для различных смещений благодаря *SIMD* инструкциям и возможность выполнять несколько инструкций одновременно на архитектуре *VLIW*.

Также можно заметить, что значения sad и $column\_sad$ для различных смещений не зависят друг от друга, поэтому их можно считать отдельно, а соответственно хранить лишь w памяти для $column\_sad$. Если хранить значения $column\_sad$ для 32-х смещений для *Full HD* изображения, то объем памяти равняется 64 КБ. Дополнительно, необходимо осуществлять доступ в векторную память к строкам двух изображений с индексами $y - s - 1$ и $y + s$, А также к строке sad и $disparity$. Таким образом, используется дополнительно 12 КБ. Итоговое количество используемой векторной памяти - 76 КБ, что укладывается в допустимое количество памяти, 128 КБ.

3.3. Описание работы алгоритма

Алгоритм заключается в следующем. Фиксируется смещение d. Вычисляется $column\_sad$ для первых $block\_size-1$ строк. Далее идет итерация по строкам. Вычисляется $column\_sad$, необходимый для определения sad текущей строки. Итерируясь по столбцам, вычисляется sad для конкретного пикселя. В зависимости от минимального значения SAD для данного пикселя и предыдущего значения смещения, обновляется минимальное значение SAD и смещение, соответствующее ему.

3.4. Векторизация

Так как значения sad и $column\_sad$ для различных смещений не зависят друг от друга, в элементах вектора можно хранить информацию, соответствующую различным смещениям, и производить операции над векторами. sad накапливается в аккумуляторном регистре типа acc24x32. Для корректных вычислений на границах используются

предикаты.

Единственный момент, когда происходит работа с несколькими смещениями сразу - поиск минимального *sad* среди различных смещений. Для этого необходимо использовать доступную в *OpenCL* функцию *reduc_ min*. Она находит минимальный элемент в векторе. Также с помощью этой функции можно реализовать *argmin* для получения смещения, зная *SAD*.

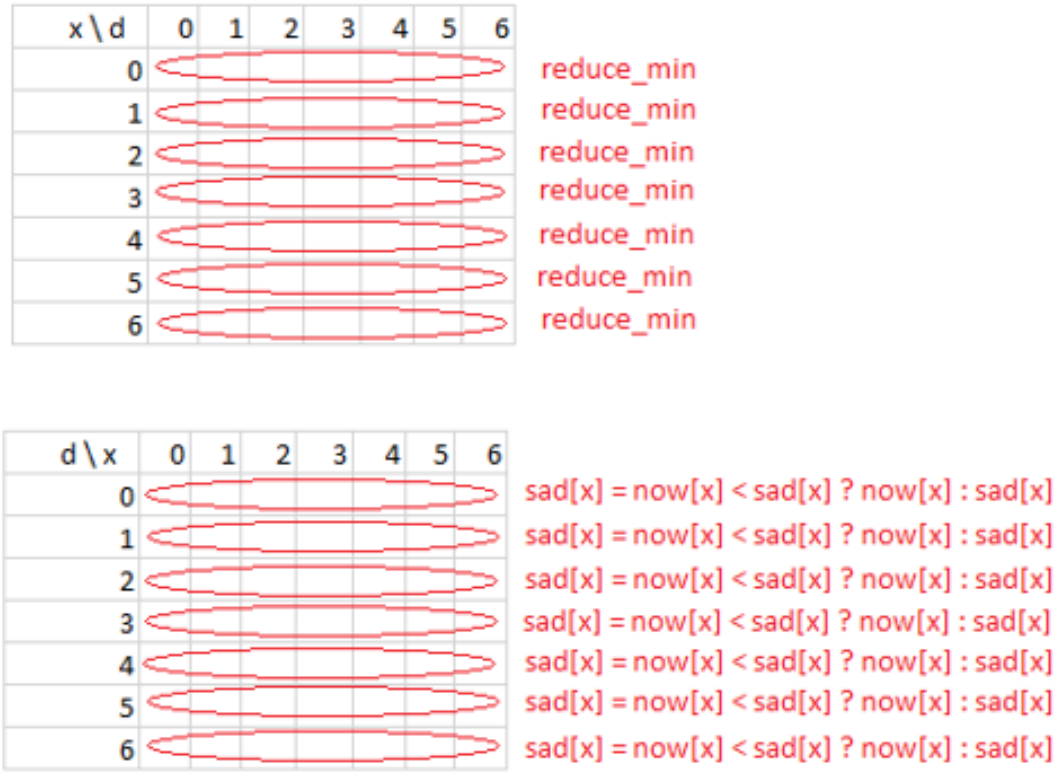

Рис. 4: Избавление от *reduc_ min*

Необходимая для этой функции инструкция не реализована на процессоре, она раскладывается в ряд других. Для вектора типа short32 она раскладывается в 9 инструкций. Это неэффективно, поэтому был найден другой способ поиска минимального значения (рис. 4).

Вместо того, чтобы непосредственно вычислить минимальное значение *SAD* с помощью функции *reduc_ min*, накопим 32 вектора для смежных пикселей в строке. Таким образом, изменив направление векторизации, вместо *reduc_ min* становится возможным использование един-

ственной инструкции *min*.

3.5. Переход от int16 к short32

Необходимо правильно рассчитать используемый тип. Для блока 21x21 (размер окна по умолчанию) таким типом является int. Но можно заметить, что минимальный *SAD* превосходит максимальное значение типа short лишь изредка. Этот замечание было проверено на тестовых данных (рис. 5). Первый график показывает, что среднее значение *sad* для конкретного смещения не превосходит 20.000, в то время как максимальное значение типа short - 65.536. При этом, максимальное значение минимального *sad* - 29.106. Таким образом, было решено использовать тип short32 для более эффективной реализации для значений размера окна не более 21.

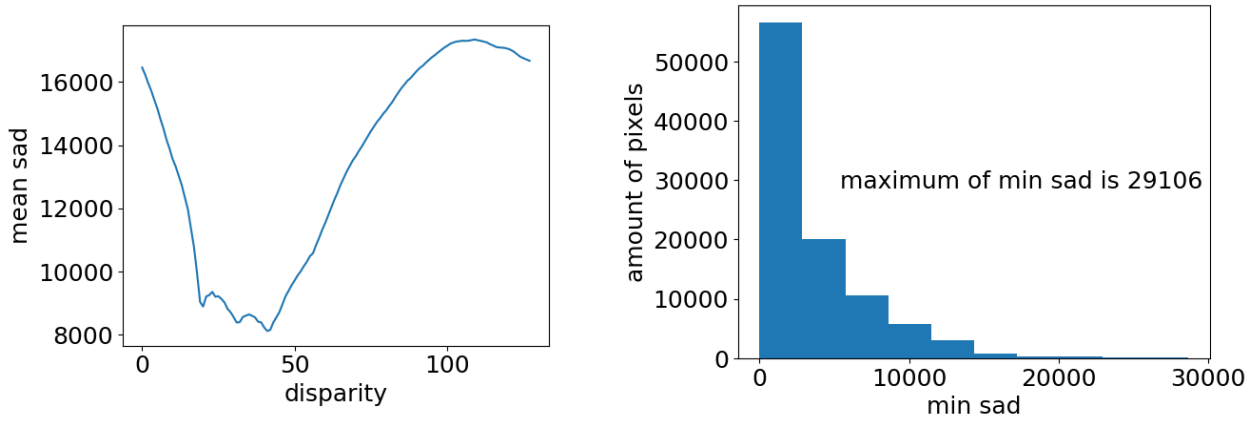

Рис. 5: На левом графике показана зависимость среднего значения *sad* в зависимости от смещения. На правом графике количество пикселей в определенном интервале минимальных значений *sad*. Количество пикселей с высокими значениями минимального *sad* мало.

4. Результаты

Было рассчитано теоретическое значение эффективности. Оно необходимо для понимания скорости работы алгоритма перед реализацией.

Некоторые типы инструкций на процессорах с архитектурой *VLIW* могут быть выполнены лишь на ограниченном количестве слотов. На процессоре *EV6x* инструкции типа *shuffle* задействуют модуль для перестановки элементов вектора, доступны только на 3-ем слоте. Инструкции типа *vmem* осуществляют операции с памятью, также доступны только на 3-ем слоте. Инструкции типа *alu*, представляющие собой арифметические операции, доступны на всех слотах. Для каждого слота отдельно считается количество тактов процессора на пиксель, и берется максимум, а также учитывается их суммарное значение. Итоговая формула:

$$E = 4 * max(mem + shuffle, (mem + shuffle + alu)/3)$$

Итоговая теоретическая оценка составляет 40 пикселей на цикл (рис. 6)

	shuffle	vmem	alu	mem+shuffle	total / 3	*Estimate, cycles per pixel*
reduce, int16	2	5	40	7	15.6	***62.4***
reduce, short32	2	5	24	7	10.3	***41.2***
no reduce, int16	4	9	14	13	9	***52***
no reduce, short32	3	7	11	10	7	***40***

Рис. 6: Теоретическая оценка

Скорость реализованного алгоритма (без *reduc_ min*, с типом short32) составляет 107 тактов процессора на пиксель. Замедление более чем в два раза можно объяснить не до конца идеальной работой компилятора с разложением *VLIW* инструкций, длительным копированием данных в векторную память, и большой загрузкой скалярного слота, который не был учтен в расчетах.

Результат работы алгоритма представлен на рис. 7.

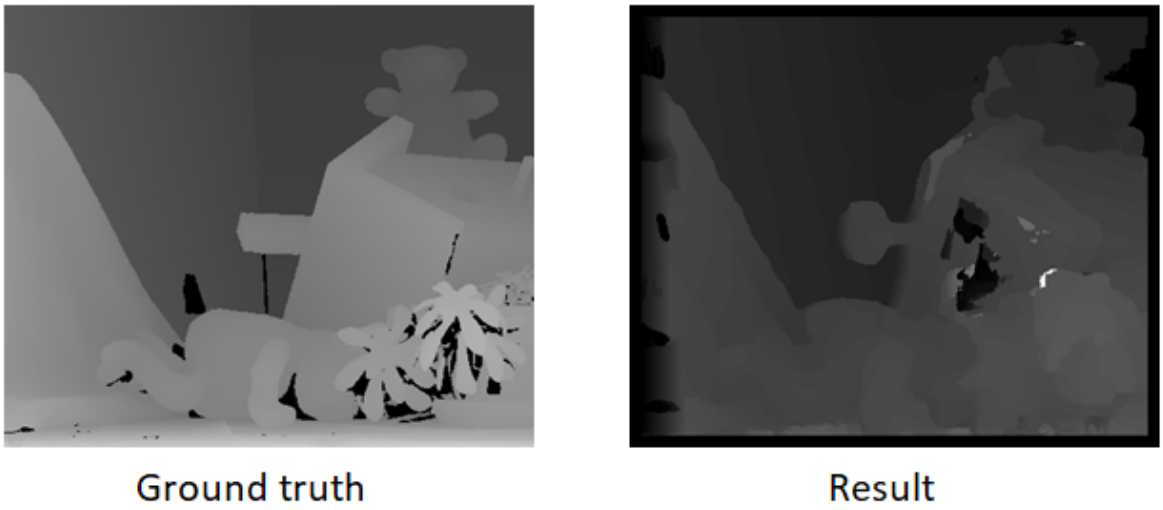

Рис. 7: Слева - истинная карта диспаритетов. Справа - карта диспаритетов, полученная алгоритмом Stereo Block Matching.

Заключение

В рамках курсовой работы были выполнены следующие задачи:

- Изучена предметная область и алгоритм, который необходимо реализовать;
- Изучена архитектура процессора;
- Построен алгоритм для данной архитектуры;
- Улучшен алгоритм:
  - Избавление от неэффективной функции;
  - Переход к типу short32;
- Оценена скорость работы и количество требуемой памяти;
- Реализован алгоритм.

В дальнейшем планируется интеграция OpenCL kernel-а в фреймворк OpenVX, который является стандартом кроссплатформенных приложений компьютерного зрения.

Список литературы

[1] Bodkin Bryan Hale. Real-Time Mobile Stereo Vision // Master's Thesis, University of Tennessee. — 2012. — Proceedings from the 15th International Conference on Vision Interface. Access mode: https://trace.tennessee.edu/utk_gradthes/1313.

[2] Brown M. Z., Burschka D., Hager G. D. Advances in computational stereo // IEEE Transactions on Pattern Analysis and Machine Intelligence. — 2003. — Aug. — Vol. 25, no. 8. — P. 993–1008.

[3] DesignWare EV6x Vision Processors. — Access mode: https://www.synopsys.com/dw/ipdir.php?ds=ev6x-vision-processors.

[4] Fisher Joseph A. Very Long Instruction Word Architectures and the ELI-512 // Proceedings of the 10th Annual International Symposium on Computer Architecture. — ISCA '83. — New York, NY, USA : ACM, 1983. — P. 140–150. — Access mode: http://doi.acm.org/10.1145/800046.801649.

[5] Hachaj T., Ogiela M. R. Real time area-based stereo matching algorithm for multimedia video devices // Opto-Electronics Review. — 2013. — Dec. — Vol. 21, no. 4. — P. 367–375. — Access mode: https://doi.org/10.2478/s11772-013-0107-5.

[6] Hirschmüller Heiko. Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information. — Vol. 2. — 2005. — 06. — P. 807–814.

[7] Patterson David A., Hennessy John L. Computer Organization and Design, Fifth Edition: The Hardware/Software Interface. — 5th edition. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2013. — ISBN: 0124077269, 9780124077263.

[8] Stefano Luigi Di, Marchionni Massimiliano, Mattoccia Stefano. A fast area-based stereo matching algorithm // Image and Vision

Computing. — 2004. — Vol. 22, no. 12. — P. 983 – 1005. — Proceedings from the 15th International Conference on Vision Interface. Access mode: http://www.sciencedirect.com/science/article/pii/S0262885604000733.

[9] Tsukuba Dataset. Middlebury Stereo Vision. — Access mode: vision.middlebury.edu/stereo/.

[10] Мокаев Руслан. Реализация алгоритма Semi-Global Matching. — 2012. — Access mode: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/445/445_Mokaev_report.pdf.

[11] Основы стереозрения. — Access mode: https://habr.com/ru/post/130300/.