Evaluation of a Feature Tracking Vision Application on a Heterogeneous Chip

Rubén Gran
University of Zaragoza
C/ María de Luna, 3. Zaragoza, 50018, Spain
e-mail: rgran@unizar.es

August Shi, Ehsan Totoni, María J. Garzarán
University of Illinois at Urbana-Champaign
201 N. Goodwin. Urbana, IL 61801, USA
e-mail: {awshi2,totoni2,garzaran}@illinois.edu

***Abstract*—Consumers of personal devices such as desktops, tablets, or smart phones run applications based on image or video processing, as they enable a natural computer-user interaction. The challenge with these computationally demanding applications is to execute them efficiently. One way to address this problem is to use on-chip heterogeneous systems, where tasks can execute in the device where they run more efficiently.**

In this paper, we discuss the optimization of a feature tracking application, written in OpenCL, when running on an on-chip heterogeneous platform. Our results show that OpenCL can facilitate programming of these heterogeneous systems because it provides a unified programming paradigm and at the same time can deliver significant performance improvements. We show that, after optimization, our feature tracking application runs 3.2, 2.6, and 4.3 times faster and consumes 2.2, 3.1, and 2.7 times less energy when running on the multicore, the GPU, or both the CPU and the GPU of an Intel i7, respectively.

***Keywords*-Evaluation of algorithms and systems, SIMD, Energy-aware systems, OpenCL.**

I. Introduction

As personal devices, such as desktops, tablets or smart phones become more powerful, the interaction between humans and computers are taking place through actions that are more natural to human users, such as gestures. These more natural interactions require the execution of applications based on vision and video analytics algorithms, which are usually highly computationally demanding, and so executing them efficiently is of the utmost importance.

High-efficiency is very important in all types of computing devices. For mobile devices, energy-efficiency is important because they operate under a very constraint power and energy-budget, as they are usually battery-operated. For desktops, reducing energy is important from an environmental perspective, but also from a budget point of view. Vision and video analytic applications are usually data parallel and adapt well to the Graphic Processor Unit (GPU). Thus, manufacturers of mobile devices are now deploying heterogeneous systems, where CPU and GPU are integrated on the same chip. Examples of those architectures include Intel Ivy Bridge [1], AMD Fusion [2], NVIDIA Tegra 250 [3], Qualcomm Snapdragon 800 [4], and Samsung Exynos 5 Octa [5], among others.

In this energy-constrained environment, both the CPU cores and GPU need to be utilized. However, it is unclear how to program these heterogeneous systems. One solution is to use OpenCL, as this programming language is supported by both the integrated GPUs and the CPU cores of these heterogeneous systems. However, it is possible that the optimizations that benefit performance when running in the GPU will hurt performance when running in the CPU, or vice versa. Also, most of the current multicores in these heterogeneous systems include vector units that can significantly speed up vision and video analytic applications. To use these vector units, OpenCL supports vector data types that the compiler maps to the underlying vector units in the computing device. These vector data types can be used even when the underlying hardware does not provide hardware support for vectorization, but it is unclear how well the code will perform.

In this paper, we explore the effectiveness of OpenCL as a unified programming paradigm for these heterogeneous systems. For that, we take a feature tracking application from the San Diego Visual Benchmark Suite (SD-VBS) [6] and evaluate its performance when running on the CPU, the GPU, and when both devices are used. We focus on a 2D-stencil or convolution computation that dominates the execution time of this application. Convolution-type computations are common in many algorithms that run on mobile systems, such as demosaic, mapping-in scale-invariant-feature-transforms (SIFT), windowed histograms, or median filtering.

We have translated the feature tracking application into OpenCL and optimized the 2D stencil computations by using built-in vector data types to take advantage of the vector units in the multicore. We have applied transformations such as loop fusion and unrolling. However, the contribution of this paper is not on the optimizations, which are well known in the compiler community, but on the performance evaluation of how effective OpenCL is as a unified programming paradigm for these heterogeneous devices. We ran experiments on an on-chip heterogeneous platform, an Intel i7 3770K with 4 cores (8 threads) with an on-chip HD4000 GPU. Our performance results show that the use of OpenCL built-in vector data types that are converted by the compiler to the appropriate instructions in the target device produced significant performance speedups, in both the multicores and the GPU.

After optimization, the 2D-stencil for a fullHD image runs up to 33.7 times faster on the multicore and 13.3 times faster on the GPU of the Intel i7. We also evaluate the performance of the overall feature tracking application when running on

the multicore, on the GPU, or when both CPU and GPU are used. Overall, our results show that the application can run 3.2, 2.6, and 4.3 times faster on the multicore, the GPU, and both, respectively. When both CPU and GPU are used, the applications consumes 2.7 times less energy.

The paper is organized as follows. Section II describes the feature tracking application and the optimizations that we apply. Section III presents our environmental setup. Section IV presents our performance results. Section V discusses other related works. Finally, Section VI concludes the paper.

II. Optimization Description

In this section, we describe the feature tracking application (Section II-A) and the optimizations that we applied (Section II-B and Section II-C).

A. Application Description

The feature tracking application from SD-VBS is based on the Kanade Lucas Tomasi (KLT) [7] algorithm. It has two main parts. The first part processes the first image to extract the initial features. The second part tracks these features across the multiple frames in the video stream, computes the movement of features, and updates the new feature positions. Profiling numbers show that the application spends 98% of the time on this second part. In this paper, we focus on this second part.

The second part of feature tracking consists of a loop that traverses the images of the video stream in sequence. In this loop, each image is processed in two phases. During the first phase, several filters are applied to each image: blurring, sobelDx and sobelDy. All these filters are instances of 2D-stencil or convolution computations. Thus, this first phase is highly parallel, since for a given filter, each pixel of an image can be processed independently of the others. The second phase compares two images to compute the movement of each feature and compute its new position. Profiling numbers show that the first phase of this second part takes 89% of the time.

Next, we explain how we transformed the feature tracking application. Section II-B describes the original 2D-stencil algorithm used in the first phase and discusses how we optimized it; Section II-C briefly describes the transformations applied to exploit the parallelism available in the second phase.

B. 2D-Stencils in Tracking

Figure 1 shows a code snippet of the 2D-stencil algorithm of feature tracking in the SD-VBS benchmark suite. All the 2D-stencil codes in this application are similar to the code in Figure 1, and only differ in the number of neighbors and the weights. As Figure 1 shows, for each pixel in the `src` image, we compute the weighted average of its 4 neighbors and store it into the `dst` image[1]. The original application traverses the image two times, the first one across the elements in a row and the second one across the elements in a column. This implementation may not be very efficient because this 2D stencil can be computed with a single pass.

[1]The division operation was replaced by the multiplication of the reciprocal

```
1
2  float src[rows][cols],dst[rows][cols],tmp[rows][cols
       ],Div = 16.0f;
3  float K[kern_size] = {4.0f,6.0f,4.0f};
4  //pass along elements in a row
5  for (int i = 0; i < rows; i++){
6    for (int j = 0; j < cols; j++){
7      tmp[i][j] = (src[i][j-1]*K[0] + src[i][j]*K[1] +
           src[i][j+1]*K[2])/Div;
8  } }
9  //pass along elements in a column
10 for (int j = 0; j < cols; j++){
11   for (int i = 0; i < rows; i++){
12     dst[i][j] = (tmp[i-1][j]*K[0] + tmp[i][j]*K[1] +
           tmp[i+1][j]*K[2])/Div;
13 } }
```

Fig. 1: 2D-stencil in feature tracking from SD-VBS

```
1  float src[rows][cols],dst[rows][cols],tmp[rows][cols
       ];
2  float4 vK[kern_size] = {{4.0f,4.0f,4.0f,4.0f},{6.0f
       ,6.0f,6.0f,6.0f},
3       {4.0f,4.0f,4.0f,4.0f}},vDiv = {16.0f,16.0f,16.0f
          ,16.0f};
4  float4 previous, next, vtmp, current, c_minus1,
       c_plus1;
5  //pass along elements in a row
6  for (int i = 0; i < rows; i++){
7    current = vload4(src[i][j]); //loads 4 float
         starting at src[i][j]
8    previous = vload4(src[i][j-4]);
9    for (int j = 0; j < cols; j+=vector_size){
10     next = vload4(src[i][j+4];
11     c_minus1[0,1,2,3] = {previous[3], current[0,1,2]};
12     c_plus1[0,1,2,3] = {current[1,2,3], next[0]};
13     vtmp = (c_minus1*vK[0] + current*vK[1] + c_plus1*
           vK[2])/vDiv;
14     tmp[i][j] = vstore4(vtmp);
15     previous = current; current = next;
16 } }
17 //pass along elements in a column
18 for (int j = 0; j < cols; j+=vector_size){
19   current = vload4(tmp[i][j]);
20   previous = vload4(tmp[i-1][j]);
21   for (int i = 0; i < rows; i++){
22     next = vload4(tmp[i+1][j]);
23     vtmp = (previous*vK[0] + current*vK[1] + next*vK
           [2])/vDiv;
24     dst[i][j] = vstore4(vtmp);
25     previous = current; current = next;
26 } }
```

Fig. 2: OpenCL 2D-stencil using vector instructions

We optimized the code using OpenCL, so that the same code can execute in the multicore and in the integrated GPU. The first optimization that we applied is the use of vector instructions. OpenCL supports built-in vector data types that are converted by the compiler to the appropriate instructions in the target device. Notice that OpenCL built-in vector data are supported even if the underlying compute device does not have hardware support.

There are two options to vectorize the loops that traverse the image along the elements in the row. One is to perform redundant loads, where the load addresses differ by the stride of the stencil. In our example this would require 3 loads of `src[i][j]`, `src[i][j-1]`, and `src[i][j+1]`. The other option does not require redundant loads, but needs shuffling instructions to move the data inter and intra registers and place each data element in the appropriate position in the vector register. The first option requires the use of unaligned

```
1  float src[rows][cols],dst[rows][cols],tmp[rows][cols];
2  float4 vK[kern_size] = {{4.0f,4.0f,4.0f,4.0f},{6.0f,6.0f,6.0
       f,6.0f},
3       {4.0f,4.0f,4.0f,4.0f}}, vDiv = {16.0f,16.0f,16.0f,16.0f
            };
4  float4 previous, next, vtmp[3], vaux, current, c_minus1,
       c_plus1;
5  //pass along elements in a column
6  for (int j = 0; j < cols; j+=vector_size){
7    previous = vload4(src[0][j-4];
8    current = vload4(src[0][j]);
9    next    = vload4(src[0][j+4]);
10   c_minus1[0,1,2,3] = {previous[3], current[0,1,2]};
11   c_plus1[0,1,2,3] = {current[1,2,3], next[0]};
12   vtmp[0] = (c_minus1*vK[0] + current*vK[1] + c_plus1*vK[2])
         /vDiv;
13   previous = vload4(src[1][j-4];
14   current = vload4(src[1][j]);
15   next    = vload4(src[1][j+4]);
16   c_minus1[0,1,2,3] = {previous[3], current[0,1,2]};
17   c_plus1[0,1,2,3] = {current[1,2,3], next[0]};
18   vtmp[1] = (c_minus1*vK[0] + current*vK[1] + c_plus1*vK[2])
         /vDiv;
19
20   for (int i = 1; i < rows; i++){
21     previous = vload4(src[i+1][j-4];
22     current = vload4(src[i+1][j]);
23     next   = vload4(src[i+1][j+4]);
24     c_minus1[0,1,2,3] = {previous[3], current[0,1,2]};
25     c_plus1[0,1,2,3] = {current[1,2,3], next[0]};
26     vtmp[(i+1)%3] = (c_minus1*vK[0] + current*vK[1] + c_plus1
           *vK[2])/vDiv;
27     vaux = (vtmp[(i-1)%3]*vK[0] + vtmp[i%3]*vK[1] + vtmp[(i
           +1)%3]*vK[2])/vDiv;
28     dst[i][j] = vstore4(vaux);
29 } }
```

Fig. 3: OpenCL 2D-stencil with vector instr. and fused loops

loads, which have a significant overhead on some architectures, such as IBM Power 7 or Intel Core 2. Unaligned loads have lower penalties in some architectures like the Intel Core i7, but they are still slower than their aligned counterparts. The option that uses shuffling instructions only uses aligned loads. The vectorization of the loops that traverse the image along the elements in a column is easier, as the data elements are in the appropriate register location and there is no need to shuffle the data or perform redundant loads. Notice that when the number of loads in an image is not a multiple of the vector size, padding might be required to guarantee that data are properly aligned. This, however, was not a problem for the images that we used, with fullHD and VGA resolution.

Figure 2 shows the vector pseudo-code using OpenCL vector data types, assuming a vector size of 4 floats and where data are shuffled. The first two loops traverse the image along the elements in a row. Each inner iteration computes the output value of 4 elements in a row. We use three vector variables (`previous`, `current` and `next`) that hold 12 consecutive elements of a row. These vectors are used to build the vectors `c_minus1` and `c_plus1`, where `c_minus1` contains the same elements as current, but shifted one position to the left, whereas `c_plus1` is shifted to the right. The inner loop `j` in line 9 only performs one load operation, and moves in steps of `vector_size`, 4 in the example. The next two loops in lines 18 and 21 of Figure 2 perform a pass through elements in the same columns. They use the vector variables: `previous`, `current` and `next` that hold 4 elements from the same columns and 3 consecutive rows.

The code in Figure 2 performs two passes through the input image, as the code in Figure 1. Thus, when images are large, the intermediate `tmp` matrix does not fit in cache. Figure 3 shows the code where a single pass through the image is required. In this figure, the fused loop traverses vectors that are in the same column. Variable `vtmp[3]` is a circular buffer that temporally holds the intermediate results of the horizontal stencil computation to compute `vaux` in line 27. The code in Figure 3 does not need the temporal image matrix `tmp[rows][cols]`. In addition, the outer loop is parallel, and no synchronization is needed between loops.

We have also unrolled the codes in Figures 2 and 3 and searched for the best number of items per OpenCL workgroup. Most previous works in the literature use loop tiling to increase locality and facilitate parallelism. This is usually applied in codes where the stencil is applied over several time steps. In the context of feature tracking, several filters are applied to the same image, and so loop tiling could be applied across the multiple filters. However, the intermediate results produced by each filter are needed, and so intermediate results still need to be saved. In this paper, we focus on the optimizations inside each filter, and we do not apply loop tiling.

C. Feature Updating

In this section, we describe the transformations we applied in the second phase of the application. This second phase consists of a loop that iterates over the valid set of features in the video stream, where the number of features can vary from dozens to hundreds, and changes during run-time. For every feature, the algorithm calculates if each feature is still valid. If it is, the algorithm updates the new position of that feature, taking into account the current frame and the previous frame. For that, the algorithm applies scaling/reduction operations in the matrix of neighbor pixels of the feature, where matrix of neighbors can be of size (8x8, 16x16, 32x32, 48x48).

In the OpenCL implementation, for the multicore CPU we wrote vector code to speed up the calculations over the small matrices of neighbors, and then we divided the problem into features and executed one of these packets into one thread-core. For the GPU, since its cores are less powerful than the CPU cores, we assigned a set of GPU-threads to each feature. Each set of threads operates over the neighbors matrix of each feature.

III. Environmental Setup

In this section, we describe the environmental setup we have used to run the experiments. Section III-A describes the platform. Section III-B discusses the software development kit and measurement methodology. Finally, Section III-C gives some extra details about the feature tracking benchmark.

A. Hardware Platform

To collect our experimental results we used an Intel i7-3770K. This processor includes an on-chip integrated Graphics Processor Intel HD4000. Characteristics of the CPU and the GPU are shown in Table I. The Intel i7 contains 4 cores and

	i7-3770K	HD4000
Clock Speed	3.4Ghz	650Mhz
Max Turbo Frequency	3.9Ghz	1.15Ghz
Peak Performance(single core)	125GFlops	147.4Gflops
Base Performance	112GFlops	44.8Gflops
# of Cores/Execution Units	4(8 threads)	16
LLC cache size	8MB	
L2 cache size	256KB	
Bandwidth to memory	25.6GB/s	25.6GB/s
RAM type	DDR3	DDR3
Instruction Set	AVX,SSE4.1/4.2	
LLC Ring connect	32B/cycle /Core	32B/cycle
MAX TDP	77W	

TABLE I: Hardware Platform characteristics [8]

supports a total of 8 threads due to hyper-threading. Each core supports AVX, which has 256-bit wide vector units and operates on 8 floating-point numbers concurrently.

The GPU in the Intel i7-3770K is the HD4000, which accesses memory through the GPU-specific L3 cache (256 KB) and the CPU and GPU Shared Last Level Cache (LLC). Accesses to global variables go through the GPU L3 cache and the LLC. Local memory (also referred as shared local memory) is allocated directly from the GPU L3 cache (which has a size of 64KB). Thus, the GPU L3 cache can be used as a scratch-pad or as a cache. For the experiments reported here, we did not use the GPU local memory. The GPU does not have AVX-like support for vectorization. In the figures in Section IV, we refer to three different platforms:

- **SC**: A single core of our Intel i7 processor. The default code executed in SC is the original C code in SD-VBS.
- **MC**: The 4 cores (8 threads) of our Intel i7 processor. Our experimental results show that our codes run faster using 8 threads instead of 4.
- **GPU**: The HD4000 GPU.

B. Development Toolkit and Measurement Methodology

Experiments were done using the Windows 7 OS. We used Microsoft Visual Studio 2010, but compiled using the Intel Composer 2014 XE and the Intel SDK for OpenCL 2013. To generate code for the multicore we used the /O3, /QAVX, /Qipo, /QPrefetch compilation flags, which by default vectorize when possible. For the Intel OpenCL compiler we used the default compilation flags.

Performance and energy metrics were collected using the Intel Performance Counter Monitor v.2.5 (PCM) driver for Windows. This driver reads the on-chip performance counters [9]. To estimate energy the hardware counters measure three domains: the consumption of the whole package (including CPU, GPU, memory controllers, etc), the consumption of the CPU domain, and the consumption of the GPU domain. For the evaluation on energy consumption in Section IV-B we report numbers for the whole package. The reason is that for this architecture, when the code is running in the multicore, the GPU does not consume energy. However, when the code runs in the GPU, the multicore is still consuming energy. This is because the CPU and the ring interconnect are in the same voltage and frequency domain [10] and the interconnect cannot be idled, since the GPU needs to access the LLC. By reporting the energy for the whole package, we are taking into account the consumption of the CPU domain when executing on GPU.

For the measurements, we launched each kernel 10 times and ran each kernel 100 times.

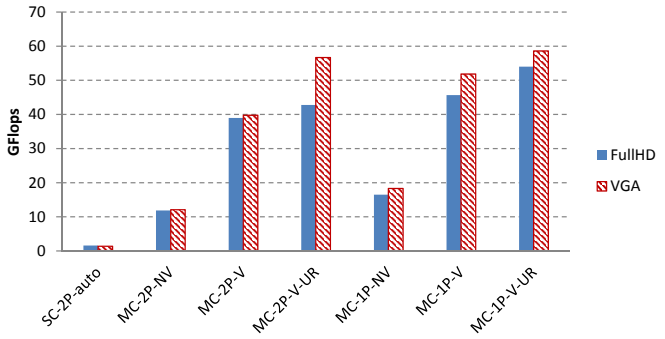

Fig. 4: Performance (GFlops) for the Intel i7 MC. SC and MC stand for the platform. 2P and 1P stand for the algorithm of Figures 2 and Figures 3, respectively. NV stands for not vectorized, V for vectorized and UR for unrolled.

C. Tracking Benchmark

As explained in Section II, we used the feature tracking application from the SD-VBS [6] that calculates the movement of a set of features over the image-flow of a video stream. We use two different image resolutions, fullHD (1920x1080) and VGA (640x480). Each fullHD image needs about 8MB of memory, which is the size of the Last Level Cache (LLC); a VGA image needs about 1MB of memory, so it fits comfortably in the LLC. The original code of this application is similar to the code shown in Figure 1. The only difference is that the image is stored as a 1D array instead of 2D, as shown in the figure. Thus, the SD-VBS benchmark access the pixel `[i][j]` using the index `[i*NumCols+j]`.

IV. Experimental Results

In this section we present our experimental results. Section IV-A shows the performance results for the optimizations in Section II; Section IV-B evaluates the energy and power dissipation; Section IV-C shows results for the whole application; Section IV-D discusses about the ease of use of OpenCL as a programming paradigm for heterogeneous platforms.

A. Performance

Multicore Performance: Figure 4 shows performance results for SC and MC. The Figure shows the GFlops per second achieved by the different code versions when running on the SC or on the MC of the Intel i7. Results are shown for two different image sizes, fullHD and VGA. The code versions executed are similar to the ones shown in Section II, but using a 2D stencil of 25 neighbors (which corresponds to the ImageBlur filter in the original feature tracking application of SD-VBS). This means that for each pixel we visit a 5x5 square matrix of pixels centered on that pixel.

`SC-2P-auto` corresponds to the original C code from SD-VBS, running in SC and compiled with /O3, which vectorizes

when possible. All the other versions are OpenCL codes that run in the MC. The bars with 2P corresponds to the code that perform two passes over the image, as in Figure 2, while the bars with 1P correspond to the code that perform one pass, as in Figure 3. NV indicates that the OpenCL code does not use the vector data type, V indicates an OpenCL code that uses the vector data type, as in Figures 2 and 3[2]. V-UR indicates that the OpenCL code uses the vector data type and has been manually unrolled. For the unrolling we tried unrolls of 2, 4, and 8 and selected the best.

The original C code SC-2P-auto performs very poorly, as the compiler was not able to vectorize. We modified this code by restricting the pointers with the restrict keyword and using the #pragma ivdep in front of the loops to tell the compiler that there are no data dependences. Then, the compiler was able to vectorize the second loop, the one that traverses the image along the elements in the columns in Figure 1. However, the compiler could not vectorize the first loop and reports that the subscript is too complex. The code with the pragmas (not shown in the figure) achieves 2.3 Gflops versus the 1.5 Gflops of the original non-vectorized code.

To assess the benefits of parallelization, we can compare the performance of SC-2P-auto versus that of MC-2P-NV. We use 8 threads, as our experiments show that 8 threads perform better than 4. Figure 4 shows that running in parallel gives us a speedup of 7.4 for fullHD (1.6 Gflops versus 11.9 Gflops). Similarly, if we compare the performance of MC-2P-NV versus MC-2P-V, we can see the benefits of vectorization. In this case, vectorization makes the code run almost 4 times faster (from 11.9 Gigaflops to 39.0 Gigaflops for fullHD). Although this is a significant improvement, it is still far from the 8 times faster that can potentially be achieved using AVX vector instructions, as they can perform 8 operations at a time. Finally, the MC-2P-V-UR bars in Figure 4 show that unrolling the vector code gives us additional benefit, as the compiler can better schedule the memory and compute instructions.

If we compare the 1P with the 2P bars we can see the benefit obtained by performing a single pass over the array. All the 1P code versions perform better than their corresponding 2P counterparts. This benefit is more noticeable for fullHD in the most optimized code version MC-1P-V-UR, as for this resolution the image does not fit in the LLC cache, and performing a single pass over the image has a higher benefit. The most optimized code versions run at 54.0 and 58.6 Gflops for fullHD and VGA, respectively. That is 43.2% and 47.6% of the peak performance.

To understand the bottlenecks that limit the performance, we ran several experiments. First, to asses the impact that limited memory bandwidth has on these algorithms we run the 1P-V-UR and 2P-V-UR algorithms with an image that fits into the L1 data caches. In this case, performance increased to 92.3 and 92.1 Gflops for 2P-V-UR and 1P-V-UR algorithms, respectively. So, when memory bandwidth is not a problem, these two algorithms perform similarly. Their performance is significantly higher than the results shown in Figure 4, reaching 73.8% of the peak performance of the machine.

Then, we noticed that the 2P algorithms require two passes over the matrix, a row-major pass and a column-major one. These passes perform the same number of floating-point operations, but have different memory access patterns. The row-major pass has a sequential memory access pattern that results in good cache locality, but requires shuffling instructions to shift elements inside the vector registers. The column major pass, does not need shuffling instructions but has a strided memory access pattern that can result in poor cache locality (a data prefetcher can predict the memory address, but needs to issue the prefetch ahead of time to hide the latency of the memory access). Thus, when we measured separately the performance of the row-major and column-major passes for the fullHD size we observed that the row-major pass of MC-2P-V-UR achieves 87.1 Gflops (69.7% of the peak) while the column-major pass only reaches 29.3 Gflops (23.4% of the peak). So, despite of the overhead of the shuffling instructions, the row-major pass runs more than twice faster than the column-major pass, due to the better memory access pattern. If we reduce the image size to fit into the L1 data cache, the performance results are just the opposite. For the L1-fitting image size, the row-major and column-major reach 84.2 and 112.6 GFLops (67.4% and 90.1% of the peak), respectively. Thus, these results show that the shuffling instructions are the bottleneck for the row-major pass to achieve the peak performance and that the performance of this row-major pass is not limited by memory bandwidth (performance is similar, 87.1 Gflops versus 84.2 Gflops, independently on whether the image fits or does not fit in the cache). However, the limited memory bandwidth is the bottleneck for the column-major pass, as this pass can achieve almost peak performance when the data fit in the cache (see Table I).

The 1P algorithms have three drawbacks: first, the memory access pattern is column-major, similar to the second pass of the 2P algorithms; second, it requires the use of shuffling instructions in order to perform vector shifts; and third, it requires reading 12 elements from each row in order to compute the stencil of only 8 elements that fit in an AVX register. Our experimental results show that the factors that limit the performance of these algorithms are the limited memory bandwidth and the overhead of the shuffling instructions. However, the most optimized OpenCL code versions produce highly efficient code, as shown by the high performance obtained when the data fits in the L1 data cache.

GPU Performance: Figure 5 shows the performance in Gflops per second obtained by the same code versions as in Figure 4 when running on SC and on the GPU of the Intel i7.

The code versions GPU-2P-V-UR in Figure 5 differ with respect to their counterparts MC-2P-V-UR in Figure 4. The reason is that in the first pass through the image in the code in Figure 2 we access the elements in the same row. However, for the GPU, it is more efficient to access the elements in the same column. Thus, we changed the code to traverse the elements along the columns, similar to the code shown in Figure 3.

[2]We use vectors of size 8 (instead of size 4 shown in Figures 2 and 3) as the Intel i7 has support for AVX that can hold 8 floats.

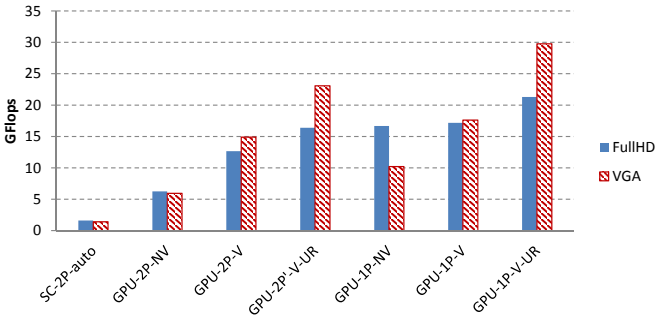

Fig. 5: Performance (GFlops) for the Intel i7 GPU. SC and GPU stand for the platform. 2P and 1P stand for the algorithm of Figures 2 and Figures 3, respectively. NV stands for not vectorized, V for vectorized and UR for unrolled

Results in Figure 5 have a similar shape as those in Figure 4. However, although the peak performance of MC and GPU are similar, the performance results are much lower for the GPU. In fact, for VGA and `1P-V-UR` (best performing version), the performance of the GPU is 29.8 Gflops (versus 58.6 in MC); for fullHD, the GPU'S performance is 21.3 Gflops (versus 54.0 in MC). The reason is that the memory hierarchy of the `MC` has better bandwidth to LLC than that of the `GPU`. While the `MC` has 4 ports to LLC and 1MB of private L2 cache, the `GPU` has a single port to LLC and 256 KB of L3 cache. In addition, GPU latencies are higher than the multicore ones, since the GPU is designed to tolerate the memory latency with large amounts of parallelism.

We have also measured the GPU performance when the image fits into the GPU L3 cache (the first level cache automatically managed in hardware like the multicore caches). In this case, we measured a performance of 36.7 Gflops for `GPU-1P-V-UR`, which is only 24.9% of the peak-performance. Since the GPU L3 latency is much higher than the multicore L1 latency, it needs larger amounts of parallelism to hide that latency, which is not available in this case. The overhead of the shuffling instructions and the need of reading 12 elements to only compute 8 are also related to the poor performance observed.

All the `1P` code versions are the same for the GPU and the CPU. Only the `2P` versions differ, as explained above. To evaluate the impact of this change, we executed the OpenCL code `MC-2P-V-UR` that we ran in the MC in the GPU, and found that in that case the code runs 26.0% slower than the `GPU-2P-V-UR` shown in Figure 5. This would be the performance penalty of using a single code version for both the CPU and the GPU. However, the most optimized version `MC-1P-V-UR` is the same for both CPU and GPU.

B. Energy Consumption and Power Dissipation

In this section, we evaluate the energy consumption and power dissipation of the different code versions when running on the two hardware platforms. Table II shows the total energy consumed (Joules) and the dissipated power (Watts). Figure 6 shows for each platform and code version, two

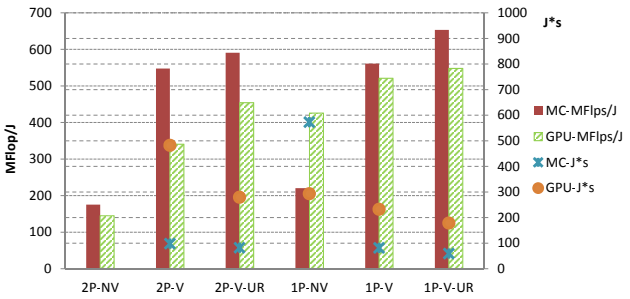

Fig. 6: MFlops/J and EDP(J*s) on fullHD. GPU and MC stand for the platform. 2P and 1P stand for the algorithm of Figures 2 and Figures 3, respectively. NV stands for not vectorized, V for vectorized and UR for unrolled.

	MC-2P			GPU-2P		
	NV	V	V-UR	NV	V	V-UR
Energy(J)	26.3	8.3	7.7	31.5	13.4	10.0
Power(W)	67.7	71.1	72.4	43.1	37.1	36.1
	MC-1P			GPU-1P		
	NV	V	V-UR	NV	V	V-UR
Energy(J)	20.7	8.1	7.0	10.7	8.7	8.0
Power(W)	74.7	81.4	82.7	39.1	33.0	38.9

TABLE II: Average Energy Consumption and Power Dissipation for fullHD for the Intel i7 per kernel. Each kernel runs 100 times.

metrics: MFlops per Joule consumed on the left-Y axis and the Energy Delay Product (EDP) [11] in Joules $\times$ Seconds on the right-Y axis. We do not plot values of EDP for the `*-2P-NV` algorithms (GPU: 2300 and MC:1000 J*s) on behalf of clarity. As discussed in Section III-B, for all the experiments, we report the energy consumed by the whole chip.

The Figure 6 shows that all code optimizations have a positive impact on these metrics. Thus, higher performance (GFlops) translates into higher energy-efficiency. The figure also shows that although the GPU is an architecture intended to leverage data-parallelism, the MC has higher Flops/Watt ratios and lower J*s than the GPU. The reason is that our implementations are not exploiting the GPU efficiently, as the performance of the GPU implementations is far from the GPU peak performance. In addition, when the codes run on the GPU, a non-negligible component of the GPU energy measured comes from the MC, since the MC and the LLC ring interconnect are in the same voltage and frequency domain. In Figure 6, the exception for this inverted behavior corresponds to `*-1P-NV`, probably because of the poor performance of this code on MC.

Table II shows that as the optimizations increase, the energy decreases. There are two reasons: first, we are better utilizing the hardware resources and second, we finish the work earlier. With respect to the dissipated power we observe that for MC, as performance increases, the dissipated power also increases (SIMD units consume more power than scalar units). However, for the GPU, energy reduces more than time, and so the power remains constant or decreases. This behavior is critical for devices limited by a power budget (battery operated).

i7-3770K	SC	GPU	MC	MC+GPU
FPS	17.0	44.4	54.4	72.4
Energy(J)	510.2	162.2	234.7	188.7
FPS/Watt	0.4	1.2	0.8	1.1
EDP	6002.3	729.8	863.8	521.6

TABLE III: Entire Tracking application results for a stream of 200 fullHD images

C. Entire Application Results

In this section, we show how the code optimizations affect the performance of the entire feature tracking application. As we presented in Section II, the tracking application kernel has two phases. In the previous subsections we have focused on the first phase (it represents 89% of time of the whole application). The second phase has also been parallelized and implemented for both MC and GPU. Table III shows the performance of the whole application running only on a single core (SC), only on the multicore (MC), only on the GPU (GPU) or using both the multicore and the GPU (MC+GPU). When running on SC or MC the GPU is idle, while when running on the GPU, the multicore is idle. However, for MC+GPU, we have implemented a pipeline where the first phase runs on the MC on one frame, and the second phase runs on the GPU and works on the results produced by the first phase (we also tested the case where the first phase runs on the GPU and the second one on MC, but the results were worse – running times increased by 46.2%). Notice that other mappings are possible and might work better for this application, but that exploration is outside the scope of this paper. SC runs the original feature tracking application from SD-VBS and compiled with /O3+auto-vect compiler flags. The rest run our most optimized code versions `1P-V-UR`.

Table III shows four rows: Frame per Second (FPS), total energy consumed by the whole chip, the ratio FPS per Watt, and the Energy Delay Product (EDP) of the total energy by the elapsed time in seconds. The table shows that all the configurations run faster than SC and that while the SC processes less than 33 FPS (real-time processing) the other three platforms (GPU, MC, MC+GPU) can process significantly more.

Results show that optimization pays off. When compared to SC, GPU, MC and GPU+MC perform 2.6, 3.2 and 4.3 times better, respectively. With respect to the energy, GPU, MC and MC+GPU consume 3.1, 2.2 and 2.7 less energy than SC.

If we consider performance and energy-consumption, the EDP metric is better for the GPU than for the CPU; MC+GPU has an even better EDP behavior than any of them individually. As Table III shows, MC+GPU processes more FPS than the others: 72.4 versus 54.4 and 44.4, while power-dissipated for every FPS is almost the same (1.1 FPS/Watt of MC+GPU versus 1.2 FPS/Watt of the best, which is the GPU). Thus, MC performs better than GPU, and MC+GPU performs better than MC. In addition, the programming effort required was not too high thanks to the OpenCL unified programming paradigm.

D. Ease of Use

OpenCL seems to be an effective programming model for the stencil computations and the feature tracking application that we studied. It allows us to write a single code version that runs well in both platforms, the multicore and the GPU, or in both platforms concurrently. Although we tuned the degree of unroll and block size differently for each platform, it is possible to use the same version for both platforms with a small penalty in performance (less than 2.3% for full HD image size and 16.8% for VGA image size). Our experimental results show that the optimizations (vectorization and loop fusion) that we applied worked well for both platforms. While we are sure that it is possible to find optimizations that work for one platform and do not work for the other one, the optimizations we applied to the feature tracking application are effective for both, MC and GPU.

To compare the performance of our OpenCL code, we wrote a C code version using AVX Intel intrinsics that implemented the code in Figure 3 and executed in a single core. We found that its performance is about 3.2 times faster than the original `SC-2P-auto`, but it is significantly slower than its counterpart `1P-V`, even discounting the effect of parallelization. Our intrinsic-based implementation is an optimized version which required a reasonable amount of effort. Thus, our experience with this experiment was that it is significantly easier to generate high-performing code with OpenCL than with intrinsics because of three reasons: first, the performance of the OpenCL code was higher; second, with OpenCL, there is no need to know the specific intrinsics of the target platform; and third, the code is portable across devices (CPU and GPU) and manufacturers. Thus, we think that the OpenCL programming paradigm can be a good option for vision applications running in these on-chip heterogeneous systems.

V. Related Work

The problem of optimizing stencil computations for multicore platforms [12], [13], [14], [15], [16] and GPUs [17], [18], [19] has extensively been studied in the past. However, those works do not evaluate the performance of an OpenCL stencil implementation that runs on a multicore and an on-chip GPU. We also analyze the performance benefit of the optimizations, in the context of whole feature tracking application where these stencils execute. The only other work that is close to ours is the work by Totoni et al [20] that also shows how to optimize an object detection application to run on a heterogeneous chip. However, their application is significantly different from ours.

Many previous works focus on stencils that solve partial differential equations, such as in climate, weather or ocean modeling [21] and where the stencil is applied for each grid point over many time steps. In this context, loop tiling has been applied to enable data locality and efficient parallelization [22], [23], [24], [25]. The works by Zhou et al. [26] and Meng et al. [17] propose the use of redundant computations to reduce the amount of synchronization when parallelizing across these timesteps. These techniques could be useful for our feature tracking application, but in our work we did not consider parallelization across the several filters applied to an image.

Previous works discuss the execution of stencil computations on the much larger discrete GPUs [19], [13]. These works

evaluate the performance impact of the thread block size, loop tiling, degree of unroll, overheads of data transfer or use of local (shared) memory. We also search the best number of items per OpenCL workgroup and considered unrolling in our experiments. For the on-chip GPU that we use and the size of our images, data transfer was not a problem. In this paper, we also evaluate the use of the OpenCL built-in vector data types when running in the CPU and the GPU.

Others propose a data layout transformation to solve the alignment and avoid the redundant loads or shuffling [16].

Heterogeneous on-chip architectures are becoming popular. Examples of those architecture include Intel Ivy Bridge [1], AMD Fusion [2], NVIDIA Tegra 250 [3], Qualcomm Snapdragon 800 [4], and Samsung Exynos 5 Octa [5]. Due to the novelty of these platforms, few studies have evaluated the performance of these applications. A recent work close to ours is the evaluation of an object removal application using OpenCL and running on a Snapdragon processor [27].

VI. Conclusions

In this paper, we evaluated the effectiveness of OpenCL as a unified programming paradigm for both the CPU and the GPU. For that, we took a feature tracking vision application and optimized it using the OpenCL built-in vector data types, loop fusion, and unrolling. Our performance numbers show that the optimized OpenCL codes run significantly faster than the non-optimized versions, and that the optimizations that were effective for one platform (multicore) were also effective for the other platform (GPU) and viceversa.

We found that the use of the OpenCL built-in vector data types and unrolling produced significant performance improvements in both, the multicore and the GPU of the Intel i7. After all optimizations, the multicore and GPU version ran 33.7 and 13.3 times (fullHD) faster than the original version. Our 25-point 2D-stencil computation can reach 54.0 and 58.6 Gflops for fullHD and VGA, respectively, when running on the multicore of an Intel i7 and 21.3 and 29.8 Gflops when running on the Intel HD4000 GPU. In all cases, performance became limited by bandwidth to some level of the memory hierarchy. In the case of the multicore, when the source image fits in the L1 cache, performance reaches 92.1 Gflops, which is 73.8% of its peak performance.

With respect to energy, we observed that the more optimized a code is, the less energy it consumes. For the multicore we observed that time reduced more than energy, which meant a higher power dissipation. For the GPU, energy reduced more than time, and so the power remained constant or decreased.

Overall, after optimization, the entire application runs 3.2 times faster when running on the multicore of the Intel i7 and 2.6 times when running on its GPU. Moreover, when both the multicore and the GPU are used, the overall feature tracking application run 4.3 times faster than the non-optimized version. In terms of EDP, the OpenCL more optimized code runs 11.5 times more efficiently than the baseline C code.

Acknowledgements

This work was supported in part by Grants TIN2010-21291-C02-01 and TIN2013-64957-C2-1-P (Spanish Government and European ERDF), gaZ: T48 research group (Aragon Government and European ESF), HiPEAC-3 NoE (European FET FP7/ICT 287759), by the NSF grant CNS 1319657, and by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign, which is sponsored by the Intel Corporation. We also thank Mert Dikmen for the help on the early stages of this work.

References

[1] S. Damaraju *et al.*, "A 22nm ia multi-cpu and gpu system-on-chip," in *Proc. of Solid-State Circuits Conference*, 2012, pp. 56–57.

[2] D. Foley *et al.*, "A low-power integrated x86-64 and graphics processor for mobile computing devices," *IEEE Journal of Solid-State Circuits*, vol. 47, no. 1, pp. 220–231, 2012.

[3] NVIDIA, "Bringing High-End Graphics to Handheld Devices," 2011. [Online]. Available: http://www.nvidia.com

[4] Qualcomm, "Snapdragon," http://www.qualcomm.com/snapdragon/.

[5] "Samsung Exynos," http://www.samsung.com/exynos.

[6] S. K. Venkata and et al., "Sd-vbs: The san diego vision benchmark suite," in *Proc. of IISWC*, 2009, pp. 55–64.

[7] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision." in *Proc. of IJCAI*, vol. 81, 1981, pp. 674–679.

[8] P. Gepner, D. Fraser, and V. Gamayunov, "Evaluation of the 3rd generation of intel core processor focusing on hpc applications," in *Proc. of PDPTA*, 2012.

[9] H. David *et al.*, "Rapl: Memory power estimation and capping," in *Proc. of ISLPED*, 2010, pp. 189–194.

[10] E. Rotem *et al.*, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, vol. 32, no. 2, pp. 20–27, March-April 2012.

[11] R. Gonzales and M. Horowitz, "Energy dissipation in general purpose microprocessors," *IEEE Journal of Solid-State Circuits*, vol. 31, no. 9, 1996.

[12] W. Augustin, V. Heuveline, and J.-P. Weiss, "Optimized stencil computation using in-place calculation on modern multicore systems," in *Proc. of Euro-Par*, 2009.

[13] K. Datta *et al.*, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proc. of SC*, 2008.

[14] H. Dursun *et al.*, "Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters," *The Journal of Supercomputing*, vol. 62, no. 2, pp. 946–966, 2012.

[15] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske, "Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization," in *Proc. of COMPSAC*, 2009, pp. 579–586.

[16] T. Henretty *et al.*, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proc. of CC*, 2011, pp. 225–245.

[17] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on gpus," in *Proc. of SC*, 2009.

[18] P. Micikevicius, "3d finite difference computation on gpus using cuda," in *Proc. of GPGPU*, 2009, pp. 79–84.

[19] A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris, "Autotuning stencil-based computations on gpus," in *Proc. of Cluster*, 2012.

[20] E. Totoni, M. Dikmen, and M. J. Garzarán, "Easy, fast, and energy-efficient object detection on heterogeneous on-chip architectures," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 45:1–45:25, Dec. 2013.

[21] A. Sawdey, M. O'Keefe, R. Bleck, and R. W. Numrich, "The design, implementation, and performance of a parallel ocean circulation model," in *Proc. of the Workshop on the Use of Parallel Processors in Meteorology*, 1994, pp. 523–550.

[22] S. Krishnamoorthy *et al.*, "Effective automatic parallelization of stencil computations," in *Proc. of PLDI*, 2007, pp. 235–244.

[23] Y. Tang, R. A. Chowdhury, and K. et al., "The pochoir stencil compiler," in *Proc. of SPAA*, 2011, pp. 117–128.

[24] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops," *TOPLAS*, vol. 26, no. 6, Nov. 2004.

[25] D. Wonnacott, "Achieving scalable locality with time skewing," *Journal of Parallel Programming*, vol. 30, p. 2002, 1999.

[26] X. Zhou *et al.*, "Hierarchical overlapped tiling," in *Proc. of CGO*, 2012, pp. 207–218.

[27] G. Wang, Y. Xiong, J. Yun, and J. R. Cavallaro, "Accelerating computer vision algorithms using opencl on mobile gpu - a case study," in *Proc. of ICASSP*, 2013.