Accelerating Applications using GPUs on Embedded Systems and Mobile Devices

Miaoqing Huang and Chenggang Lai
Department of Computer Science and Computer Engineering
University of Arkansas
Fayetteville, AR 72701, USA
Email: {mqhuang,cl004}@uark.edu

Abstract—Graphics processing units (GPUs) are capable of achieving remarkable performance improvements for a broad range of applications. However, they have not been widely adopted in embedded systems and mobile devices as accelerators mainly due to their relatively higher power consumption compared with embedded microprocessors. In this work, we conduct a comprehensive analysis regarding the feasibility and potential of accelerating applications using GPUs in low-power domains. We use two different categories of benchmarks: (1) the Level 3 BLAS subroutines, and (2) the computer vision algorithms, i.e., mean shift image segmentation and scale-invariant feature transform (SIFT). We carried out our experiments on the Nvidia CARMA development kit, which consists of a Nvidia Tegra 3 quad-core CPU and a Nvidia Quadro 1000M GPU. It is found that the GPU can deliver a remarkable performance speedup compared with the CPU while using a significantly less energy for most benchmarks. Further we propose a hybrid approach to developing applications on platform with GPU accelerators. This approach optimally distributes workload between the parallel GPU and the sequential CPU to achieve the best performance while using the least energy.

I. INTRODUCTION

Graphics processing units (GPUs) are capable of achieving remarkable performance improvements for a broad range of applications. They have been adopted in many supercomputers, e.g., Titan, Stampede, and Tianhe-2, mainly for two purposes: (1) improving the performance, and (2) reducing the overall power consumption [1]. As GPUs are becoming ubiquitous in HPC, numerous applications have been ported to GPU-based systems over the past several years, including large scale scientific applications on GPU clusters [2]–[4].

Although being successful in high-performance domains, most current GPU devices cannot find their places in embedded systems and mobile devices. Embedded systems and mobile devices typically have a quite tight power budget, particularly for hand-held devices and smart phones that are powered by batteries. Originally, the embedded systems and mobile devices were designed for dealing with targeted tasks and therefore had application-specific architectures. As these devices become pervasive, the computation demanding is increasing. More sensors and high-resolution cameras are included in smart phones. Digital processing and image processing capabilities are essential for analyzing data to make real-time decisions.

In this work, we study the potential use of GPUs in embedded systems and mobile devices. We execute benchmarks such as Level 3 BLAS subroutines as well as computer vision algorithms on the Nvidia CARMA development kit, which contains both low-power multicore CPU and low-end GPU. Although the existing GPU devices are still too power-demanding for embedded systems and mobile devices, the results of this paper demonstrate that GPUs are capable of achieving more than one order of magnitude performance speedup for sophisticated computer vision algorithms while using significantly less energy compared with CPUs. Based on the results in this work, we expect that future embedded systems and mobile devices will become hybrid systems in which the CPU deals with most tasks flexibly while GPUs are used to handle most parallel tasks.

The remainder of this paper is organized as follows. The related work is briefly discussed in Section II. Section III introduces the methodology used in this work. The benchmarking results and the analysis are presented in Section IV. Finally, some conclusion marks are given in Section V.

II. RELATED WORK

Hardware accelerators have been widely used in embedded system to improve the performance under power budget. Ouellette and Connors introduced a hardware floating-point unit (FPU) and a hardware DCT core on Xilinx Virtex-II FPGAs to offload software subroutines [5]. Several hardware accelerators for embedded media applications, such as MP3 and H.264 decoders, are introduced in [6]. Altera showcased some hardware accelerators that can improve the performance and save power for Mandelbrot algorithm [7]. A co-design method for embedded system with hardware acceleration is introduced in [8]. Digital signal processor (DSP) is another special microprocessor widely used in embedded system for performance improvement. For instance, many applications have been showcased on such systems with DSPs, including Bokeh application [9], numerical control [10], image recognition [11], etc. A lot of work have been proposed to improve the performance of the applications running on embedded DSP systems from the perspectives of code optimization [12], [13], memory architecture [14] among others. However, each hardware accelerator is only able to carry out a specific function. Therefore they may be not a good solution for general-purpose platforms, such as mobile devices. Although DSPs can be used to implement a broader range of applications compared with hardware accelerators, they are specialized for digital signal processing. Further, the performance speedup by using DSPs is quite insignificant compared with accelerators.
Recently, several work [15]–[17] implemented image processing algorithms such as face recognition and SIFT on mobile GPUs, e.g., Nvidia Tegra [18], Adreno [19], and PowerVR SGX [20]. These GPUs are integrated on the same system-on-chip with the CPUs. Their computing capability is quite limited compared with those standalone GPUs such as the one used in this work. For example, the image size in [17] is $320 \times 240$, which is way below the typical resolution of cameras equipped on today’s mobile systems.

In order for the GPU to be adopted as a general-purpose hardware accelerator in embedded systems and mobile devices, only to keep $\frac{\text{performance speedup}}{\text{energy consumption growth}} \approx 1$ is not acceptable. For example, a GPU that is able to reduce the computation time by $40\times$ while consuming 40 times energy cannot meet the power budget in almost all embedded systems. To be attractive, GPUs need to provide remarkable performance speedup while using the same or even much less energy. The results in this work show that GPUs are capable of achieving both performance improvement and energy saving.

III. METHODOLOGY

We adopted an empirical approach in this work. The experiments were carried out on a CARMA MXM development kit from Nvidia [21], as shown in Fig. 1(a). The CARMA kit contains one Nvidia Tegra 3 quad-core CPU [18] and one Nvidia Quadro 1000M GPU. Both the Tegra 3 CPU and Quadro 1000M GPU are equipped with 2 GB memory. The Tegra 3 processor consists of 4 ARM Cortex A9 high-performance cores and 1 low power, low leakage “Battery Saver” core, which is also an ARM coretex A9 core. The 4 high-performance cores run at 1.3 GHz. The Quadro 1000M is based on Nvidia’s Fermi GPU architecture [22] and contains 96 CUDA cores running at 700 MHz. The Quadro 1000M GPU contains 3 streaming multiprocessors (SMs), each of which contain 32 processing cores (i.e., CUDA cores) as shown in Fig. 1(b). Each CUDA core is 32-bit. Two CUDA cores can combinedly deal with a 64-bit double-precision floating-point operation.

![CUDA GPU Tegra ARM CPU](image)

(a) CARMA development kit.

Fig. 1. Nvidia’s CARMA development kit with Fermi-based GPU.

The CARMA board is pre-installed with an OS of Ubuntu 11.04. The CUDA installed on the board is version 4.2. An application has to be first cross-compiled on a Linux box with the same OS and CUDA. Then the executable is copied to the board for benchmarking.

We have up to four different types of implementation for each benchmark as follows.

- **Sequential implementation on CPU.** The implementation is a single-thread application running on a single ARM core.
- **Parallel implementation on CPU.** Pthreads is used to parallelize the workload so that the computation is carried out on 4 ARM cores.
- **Parallel implementation on GPU.** All the computation is carried out by the GPU processor.
- **Hybrid implementation.** The sequential part of a program runs on the CPU and the parallel part runs on the GPU.

We used a Kill A Watt P4400 Electricity Load Meter and Monitor by P3 International [23] to measure the power consumption of the CARMA board. We first measured the power consumption when both CPU and GPU were idle, i.e., the static power consumption. We measured the power consumption again when application was running on a processor. This power consumption is called dynamic power consumption. The difference between the dynamic power consumption and the static power consumption is the power spent on the application by the processor. We found the static power consumption of the CARMA board is $\sim 13$ W. The typical power consumption of an ARM core is $\sim 2$ W. The maximum power consumption of the Quadro 1000M GPU is $\sim 45$ W. The GPU consumes much less power for most benchmarks in this work.

IV. BENCHMARKING RESULTS

We tried to port the traditional benchmarks, such as CoreMark [24] and ParMiBench [25], onto the GPU. However, it was found that the parallelism in those benchmarks was not high enough to achieve reasonable performance speedup on GPU. Therefore, we do not report the detailed result for CoreMark and ParMiBench in this work. Instead, we choose the BLAS (Basic Linear Algebra Subroutine) and computer vision algorithms. BLAS is used in numerous applications, therefore, has a board impact. Computer vision algorithms are critical for embedded systems and mobile devices to deal with images for applications such face recognition and image registration.
TABLE I. Performance speedup and energy saving on GPU for BLAS subroutines.

<table>
<thead>
<tr>
<th>BLAS Subroutine</th>
<th>Performance Speedup</th>
<th>Energy Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU (s)</td>
<td>GPU (s)</td>
</tr>
<tr>
<td>dgemm</td>
<td>1584.51</td>
<td>6.36</td>
</tr>
<tr>
<td>dsymm</td>
<td>950.49</td>
<td>6.63</td>
</tr>
<tr>
<td>syyrk</td>
<td>781.93</td>
<td>3.31</td>
</tr>
<tr>
<td>strmm</td>
<td>837.35</td>
<td>3.32</td>
</tr>
<tr>
<td>dtrsm</td>
<td>906.17</td>
<td>3.52</td>
</tr>
</tbody>
</table>

*CPU power consumption: 2 W.
†: Matrix size: 4,096\times4,096.

A. BLAS

BLAS is widely used in scientific applications. In this work we carried out 5 Level 3 subroutines as follows.

- **xGEMM**: general matrix-matrix operation, i.e., $C \leftarrow \alpha \times A \times B + \beta \times C$, where $\alpha$ and $\beta$ are scalars, and $A$, $B$ and $C$ are matrices.
- **xSYMM**: matrix-matrix operation, i.e., $C \leftarrow \alpha \times A \times B + \beta \times C$, where $A$ is a symmetric matrix.
- **xSYRK**: matrix-matrix operation, i.e., $C \leftarrow \alpha \times A \times A' + \beta \times C$, where $C$ is an $n \times n$ symmetric matrix and $A$ is an $n \times k$ matrix.
- **xTRMM**: matrix-matrix operation, i.e., $B \leftarrow \alpha \times \text{op}(A) \times B$ or $B \leftarrow \alpha \times B \times \text{op}(A)$, where $\text{op}(A) = A$ or $\text{op}(A) = A'$.
- **xTRSM**: solves one of the matrix equations, i.e., $B \leftarrow \alpha \times B$ or $X \times \text{op}(A) = \alpha \times B$, where $A$ is a triangular matrix, $\text{op}(A)$ is either $\text{op}(A) = A$ or $\text{op}(A) = A'$.

There are 3 implementations for these subroutines, i.e., sequential CPU implementation, parallel CPU implementation, and GPU implementation.

- The sequential CPU implementation is based on the library CLAPACK 3.2.1 [26]. The corresponding Level 3 subroutines in the CLAPACK library are directly called to carry out the computation.
- The parallel CPU implementation is based on Pthreads and the CLAPACK library. A large matrix $A$ is decomposed into 4 blocks, i.e., $A_{00}$, $A_{01}$, $A_{10}$, and $A_{11}$. The operations between two matrices, $A$ and $B$, are implemented by the operations between $A_{ij}$ and $B_{mn}$. The workload is evenly distributed among the 4 ARM cores.
- The GPU implementation is based on the CUBLAS library. The Level 3 subroutines in the CUBLAS library are directly called for computation.

We first perform a simple comparison between the single-thread CPU implementations and the corresponding GPU implementations of all five subroutines on 4,096\times4,096 matrices.

The operation time and the power consumption of the processors for various cases are recorded in Table I. It is found that a single ARM core consumes constantly $\sim 2$ W power during the operation. The power consumption of the GPU varies for different applications due to varying device occupancy. ARM Cortex A9 is a 32-bit core. Therefore, the double-precision performance of the ARM core is roughly half of the single-precision performance. To operate on matrices of the same size, the double-precision implementation roughly takes twice the time of the single-precision counterpart. On the other hand, Quadro 1000M GPU is a device mainly targeting consumer market. Its double-precision capability is much lower than the single-precision. This is the reason why the DGEMM subroutine takes more than 6 times the time than the SGEMM subroutine. Overall, the GPU implementation is capable of providing hundreds to thousands of performance speedup while consuming a significantly less energy, i.e., averagely $\sim \frac{1}{10}$ for single-precision and $\sim \frac{1}{20}$ for double-precision, compared with the sequential CPU implementation.

In order to get a sense how the performance and power consumption vary as the problem size changes on both processors, we carry out an experiment on xGEMM for 5 different matrix sizes as listed in Table II. Based on the results, it can be found that both CPU and GPU are very efficient in dealing with small matrices. When the matrix size is small, say, 256\times256, the on-board cache can accommodate all the data, resulting in higher performance and less power consumption. When the matrix size increases, the performance degrades mainly...
TABLE II. PERFORMANCE SPEEDUP AND ENERGY SAVING ON xGEMM FOR MATRICES OF VARYING SIZES.

<table>
<thead>
<tr>
<th>Matrix Sizes</th>
<th>CPU (ms)</th>
<th>GPU (ms)</th>
<th>Speedup</th>
<th>GPU (J)</th>
<th>Power Consumption (W)</th>
<th>Energy Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.29</td>
<td>1.16438</td>
<td>0.68</td>
<td>6.7</td>
<td>0.002</td>
<td>347.58</td>
</tr>
<tr>
<td>512</td>
<td>2.10</td>
<td>7.0342</td>
<td>2.96</td>
<td>7.7</td>
<td>0.02</td>
<td>182.71</td>
</tr>
<tr>
<td>1,024</td>
<td>16.34</td>
<td>887.23</td>
<td>59.00</td>
<td>21.2</td>
<td>0.35</td>
<td>83.70</td>
</tr>
<tr>
<td>2,048</td>
<td>129.65</td>
<td>892.82</td>
<td>23.50</td>
<td>36.4</td>
<td>4.72</td>
<td>49.06</td>
</tr>
<tr>
<td>4,096</td>
<td>1,035.23</td>
<td>914.19</td>
<td>0.94</td>
<td>36.6</td>
<td>37.89</td>
<td>49.96</td>
</tr>
</tbody>
</table>

TABLE III. PERFORMANCE SPEEDUP AND ENERGY SAVING ON DSGEMM BY PARALLEL CPU IMPLEMENTATION.

<table>
<thead>
<tr>
<th>Matrix Sizes</th>
<th>S-CPU (s)</th>
<th>P-CPU (s)</th>
<th>Speedup</th>
<th>S-CPU (W)</th>
<th>P-CPU (W)</th>
<th>Power Consumption (W)</th>
<th>Energy Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>0.53</td>
<td>0.34</td>
<td>1.56</td>
<td>1.06</td>
<td>1.34</td>
<td>0.44</td>
<td>2.40</td>
</tr>
<tr>
<td>512</td>
<td>2.85</td>
<td>0.89</td>
<td>3.09</td>
<td>5.70</td>
<td>3.2</td>
<td>2.86</td>
<td>1.99</td>
</tr>
<tr>
<td>1,024</td>
<td>24.21</td>
<td>8.14</td>
<td>2.97</td>
<td>48.42</td>
<td>4.5</td>
<td>36.65</td>
<td>3.32</td>
</tr>
<tr>
<td>2,048</td>
<td>196.83</td>
<td>59.59</td>
<td>3.30</td>
<td>393.66</td>
<td>4.5</td>
<td>268.15</td>
<td>1.47</td>
</tr>
<tr>
<td>4,096</td>
<td>1,584.51</td>
<td>522.12</td>
<td>3.03</td>
<td>3,169.02</td>
<td>4.5</td>
<td>2,349.55</td>
<td>1.35</td>
</tr>
</tbody>
</table>

*CPU power consumption: 2 W.
†: All matrices are square matrices. Only the size along one dimension is listed.
‡: S-CPU: sequential CPU implementation.
§: P-CPU: parallel CPU implementation consisting of 4 threads.

Fig. 3. Performance speedup and energy saving on xGEMM for matrices of varying sizes.

due to the increasing cache misses and the higher memory access overhead. The increased cache and memory activities contribute to the higher power consumption as well. As shown in Fig. 3, both performance speedup and energy saving start stabilizing as the matrix size reaches $1,024 \times 1,024$.

Since there are 4 cores on the ARM processor, it is worthwhile to evaluate how much performance speedup and energy saving can be achieved by the parallel implementation on multicore CPUs. We use Pthreads to parallelize the DSGEMM implementation on the 4 cores. A large matrix $A$ is decomposed into 4 blocks, i.e., $A_{00}$, $A_{01}$, $A_{10}$, and $A_{11}$. The matrix-matrix operations on the matrix blocks, i.e., $A_{ij}$ and $B_{mn}$, are handled with by calling the Level 3 subroutines in the CLAPACK library. The results of the parallel CPU implementation are shown in Table III. Averagely, the parallel implementation involving 4 cores can achieve a $\sim 3 \times$ speedup due to the synchronization overhead for multiple threads. The power consumption of 4 cores is increased to $\sim 4.5$ W when the matrix size is $\geq 1,024 \times 1,024$. As a result, the energy saving from the parallel CPU implementation is very limited, i.e., roughly 30%. By comparing the results in Table II and Table III, it can be found that GPU is at least one order of magnitude more efficient than multicore CPU for both performance speedup and energy saving.

B. Computer Vision Algorithms

Computer vision is a field that includes methods for acquiring, processing, analyzing, and understanding images. We used two representative algorithms, mean shift segmentation [27], [28] and scale-invariant feature transform (SIFT) [29].

1) Mean Shift Segmentation: Segmentation is the process of partitioning a digital image into multiple segments to simplify and/or change the representation of an image into a fashion that is more meaningful and easier to analyze. Image segmentation is typically used to locate objects and boundaries (lines, curves, etc.) in images [30]. The result of image segmentation is a set of segments that collectively cover the entire image, or a set of contours extracted from the image.

Mean shift considers feature space as an empirical probability density function. If the input is a set of points then mean shift considers them as sampled from the underlying probability density function. If dense regions are present in the feature space, then they correspond to the local maxima of the probability density function. If each data point, mean shift associates it with the nearby peak of the data set’s probability density function. For each data point, mean shift defines a window around it and computes the mean of the data point. Then it shifts the center of the window to the mean and repeats the algorithm till it converges. After each iteration, we can consider that the window shifts to a more denser region of the data set. At the high level, we can specify mean shift as follows:

1) Define a window around each data point;
2) Compute the mean of data within the window;
3) Shift the center of the window to the mean and repeat till convergence, i.e., the center of the window no longer shifts.

This process is illustrated in Fig. 4 in which particles are used as an example. The original position of the window is shown in Fig. 4(a). The geometric center of the window (i.e., $GC_1$) does not overlap with the center of the mass (i.e., $MC_1$). Therefore, the geometric center of the window is shifted to the center of the mass (i.e., $GC_2 \leftarrow MC_1$) in Fig. 4(b). Then a new center of the mass is calculated (i.e., $MC_2$). If the geometric center and the mass center do not overlap, the geometric center will keep shifting until these two centers overlap (i.e., $GC_n = MC_n$), as shown in Fig. 4(d).

In image segmentation, each pixel is dealt with as a data point. The coordinate of the pixel is initially set as the geometric center of the window. Then a center of pixel intensity is calculated using the pixels within the window. The geometric center of the window will shift to intensify center of the window until these two centers overlap, i.e., the convergence is reached.

![Image 5](attachment:image5.png)

Fig. 5. Image segmentation on a spaceborne image using mean shift.

![Image 6](attachment:image6.png)

Fig. 6. Mean shift result.

We tested the mean shift algorithm on spaceborne globe images of various resolutions. The globe image with resolution $640 \times 640$ and its segmentation are shown in Fig. 5. The clouds are difficult to be distinguished from the oceans and the lands in grayscale images. However, as shown in Fig. 5(b), mean shift algorithm is capable of providing a meaningful segmentation by separating lands, oceans, and clouds.

The implementation of mean shift segmentation consists of two steps, mean shift filtering and region fusion. It is noticed that the time spent on the region fusion step is negligible compared with the mean shift filtering step when the image size is $\leq 2,048 \times 2,048$. Therefore, we only report the results on the step of mean shift filtering.

Given a pixel $P_n$ in the source image and a search window with radius $h_s$, the mean shift process is to repeatedly calculate the mean shift vector $\Delta h$ (shown in Fig. 4) until the squared magnitude of $\Delta h$, i.e., $\|\Delta h\|^2$, is less than a threshold $\epsilon$. The number of elements in vector $\Delta h$ can be 5 if dealing with color images, i.e., the change of $x$ coordinate ($\Delta x$), the change of $y$ coordinate ($\Delta y$), and the changes of intensities in RGB ($\Delta R$, $\Delta G$, $\Delta B$). For grayscale image, only three elements are needed for $\Delta h$, i.e., $\Delta x$, $\Delta y$, and $\Delta I$. If we use $S$ to denote the window centered at $P_n$, and use $s$ to denote a pixel within the window, $\Delta x$ can be calculated as (1), in which $I_s$, $x_s$, $w_s$, and $h_s$ are the intensity, the $x$ coordinate, the weight of pixel $P_s$, and the range bandwidth, respectively. The weight of each pixel in an image is specified by a predefined weight map.

$$\Delta x = \sum_{s \in S} K \left( \frac{x_s - h_s}{h_r} \right) w_s x_s,$$

where $K(x) = \begin{cases} 1 & \text{if } \|x\| < 1 \\ 0 & \text{if } \|x\| \geq 1 \end{cases}$

![Image 7](attachment:image7.png)

Fig. 7. Mean shift filtering.

![Image 8](attachment:image8.png)

Fig. 8. Region fusion.

![Image 9](attachment:image9.png)

Fig. 9. Performance speedup and energy saving on GPU for mean shift filtering.

<table>
<thead>
<tr>
<th>Image Size</th>
<th>CPU (s)</th>
<th>GPU (s)</th>
<th>Speedup</th>
<th>Energy CPU (J)</th>
<th>GPU (J)</th>
<th>Saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>240x240</td>
<td>24.36</td>
<td>4.02</td>
<td>6.06</td>
<td>48.72</td>
<td>80.40</td>
<td>0.61</td>
</tr>
<tr>
<td>320x320</td>
<td>43.70</td>
<td>6.30</td>
<td>6.94</td>
<td>87.40</td>
<td>126.00</td>
<td>0.69</td>
</tr>
<tr>
<td>500x500</td>
<td>99.87</td>
<td>12.96</td>
<td>7.71</td>
<td>199.74</td>
<td>259.20</td>
<td>0.77</td>
</tr>
<tr>
<td>640x640</td>
<td>168.71</td>
<td>20.55</td>
<td>8.21</td>
<td>337.42</td>
<td>411.00</td>
<td>0.82</td>
</tr>
<tr>
<td>1024x1024</td>
<td>478.91</td>
<td>49.72</td>
<td>9.63</td>
<td>957.82</td>
<td>994.40</td>
<td>0.96</td>
</tr>
<tr>
<td>2048x2048</td>
<td>1,841.42</td>
<td>186.23</td>
<td>9.89</td>
<td>3,682.84</td>
<td>3,724.60</td>
<td>0.99</td>
</tr>
</tbody>
</table>

*Power consumption: CPU: 2 W; GPU: 20 W.
\( \Delta y \) and \( \Delta I \) are computed in a similar way. A new center of the window \( P_b \) is calculated as \( P_b = P_a + \hat{M}_h \). Then the mean shift vector to \( P_b \) is calculated until the center of the window no longer shifts.

The above computation in the mean shift filtering on a pixel is implemented in a GPU kernel function. When the kernel is launched, each pixel in the source image is handled by a GPU thread. If the size of image is \( m \times n \times n \) threads are created and scheduled to execute on hundreds of processing cores on a GPU device.

The pure software implementation of the mean shift algorithm is adapted from the open source code developed by the Robust Image Understanding Lab at Rutgers University and is based on papers [28], [31]. The software implementation is a single-thread one. We manually converted the mean shift filtering step to the CUDA C for running on the GPU. The GPU implementation of the mean shift filtering process contains only one GPU kernel function, which is responsible for generating one pixel in the filtered image. Therefore the number of GPU threads is same to the number of pixels in the original image. These threads are grouped into 1-dimensional thread blocks, each of which contains 128 threads. Due to the improved memory hierarchy on Fermi GPU architectures, typical performance optimization techniques, such as memory coalescing and memory prefetching [32], are not implemented in the mean shift filtering.

The performance and power consumption of both CPU and GPU implementations are shown in Table IV and Fig. 6. The power consumptions of both CPU and GPU are quite stable, i.e., \( \sim 2 \text{ W} \) and \( \sim 20 \text{ W} \), respectively, for various image size. It can be found that the performance improvement on GPU climbs as the size of image increases. For the largest image, i.e., \( 2,048 \times 2,048 \), the GPU is almost 10 times faster than the CPU while consuming the same amount of energy. When the image resolution increases, more GPU thread blocks are available to be scheduled to stream multiprocessors, resulting in a higher GPU occupancy and a better performance. Once the occupancy is maximized, adding more thread blocks cannot further increase the processing throughput.

2) Hybrid Design of SIFT Algorithm: SIFT is an algorithm in computer vision to detect and describe local features in images. The whole SIFT process is divided into 8 stages as shown in Fig. 7. Each of the 8 stages has its own degree of intrinsic parallelism. Therefore, a hybrid design involving both CPU and GPU may be more appropriate than the implementation on a single device. We followed an empirical approach to figuring out the optimal distribution of workload between the CPU and the GPU.

We first downloaded the C++ implementation of SIFT algorithm by Dr. Andrea Vedaldi from http://www.vlfeat.org/~vedaldi/code/siftpp.html. This implementation is a single-thread version. The pure software implementation is first applied to the grayscale image shown in Fig. 8(a). The size of the image is \( 4,288 \times 2,848 \), the typical image resolution used by most digital cameras and smart phones. The performance and the power consumption of the 8 stages are listed in Table V. It is found that the power consumption of most stages is \( \sim 2.5 \text{ W} \). In the 8 stages, most of the time is spent at 4 stages, i.e., convolution, DoG, octave gradient, and key description generation. In the second step, we converted the whole SIFT algorithm to its GPU version and applied the GPU implementation to the same image. As expected, the computation time for those 4 stages is significantly reduced. In the meantime, the power consumption of GPU is not necessarily much higher than the CPU for most stages except the octave gradient stage. Based
on these observations, we decided to allocate the 4 most time-consuming stages to GPU and leave the other 4 stages to CPU in the hybrid design, as shown in Fig. 7. Although it can save 7 joules by keeping the octave gradient stage on CPU, it is not worthwhile to increase the time by 45% for saving only 3.4% energy.

By looking at the total operation time and power consumption of the three implementation, it can be found that the pure GPU implementation alone can significantly reduce the computation time by more than one order of magnitude. The hybrid design can further reduce the total time slightly and cut the power consumption by another 22%.

It is also worth mentioning that the above three implementations generate almost the same number of key points, i.e., ̃179,000. These key points are illustrated as blue vectors in Fig. 8(b).

V. CONCLUSION

GPU has been widely adopted as a hardware accelerator in high-performance domain for performance improvement and energy saving. In this work we demonstrate the encouraging potential for adopting GPU in embedded systems and mobile devices as accelerators. By deploying various benchmarks on a hybrid board, which consists of a low-power embedded microprocessor and a low-end GPU, our results show that the GPU can achieve at least one order of magnitude performance speedup for both Level 3 BLAS subroutines and complex computer vision algorithms while using the same or much less energy compared with the CPU. We further demonstrate that on such hybrid systems containing both CPU and GPU, a careful distribution of workload between these two types of processors is essential for obtaining the best performance while consuming the least energy.

ACKNOWLEDGMENT

This work is partially supported by NASA grant 12-2012EPSCoR-0026. The authors would like to thank Nvidia Corporation for the donation of CARMA development kit.

REFERENCES


