Offloading to the GPU: An Objective Approach

Ajaykumar Kannan Mario Badr Parisa Khadem Hamedani Natalie Enright Jerger
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
{kannanaj, badrmari, parisa, enright}@ece.utoronto.ca

1. Introduction

Certain parts of applications exhibit high amounts of parallelism in terms of threads, data, and instructions. The recent shift to multi-threaded applications is becoming ubiquitous, and developers have the option of running threads on CPUs or GPUs. Offloading these computations to a GPU can improve performance and energy consumption; when running at high throughput, GPUs inherently perform better and are more energy efficient than CPUs [3, 4]. Unfortunately, not all computations offloaded to a GPU translate into better performance and energy. Simply porting an entire application to a GPU will not improve performance by an order of magnitude because GPUs rely on simple cores. Deciding which parts (or threads) of an application should be ported to use the GPU is not a menial task. A number of factors need to be taken into consideration, such as the working set size, the length of the kernel, and the kernel's instruction mix. If these factors are not considered, naïvely porting computations to the GPU may result in a net loss of performance and energy.

In this paper, we propose a systematic methodology to determine which parts of a CPU application should be ported to a GPU. We narrow our study to mobile platforms where battery life (i.e. energy consumption) is of paramount importance. We rely on previous models that provide visual [3, 6, 10] and analytic [3, 4] information regarding performance, energy, and parallelism. At a high level, our methodology uses regression coefficients that tell us the *cost* (time and energy) of an operation for a given system (Section 2). Then, profiling an application at the CPU level will determine the instruction mix and parallelism in different regions of the code (Section 3). Combining the application profile with the regression coefficients, we can objectively compare how a region of code will perform on two different systems (i.e., the CPU and GPU).

2. Modelling the CPU and GPU

Applications and algorithms can be characterized as having an operational intensity: the ratio of *useful operations* to the working set size (in number of bytes) [10]. We define *useful operations* as non-control flow operations, namely arithmetic and memory operations. Depending on the intensity of a workload there is an upper bound on the performance [10] and energy [3] that an architecture can deliver, referred to as the *roofline*. Figure 1 shows the empirical rooflines for the CPU (Krait 450) and GPU (Adreno 420) of a Qualcomm Snapdragon 805 [1] with performance in terms of floating

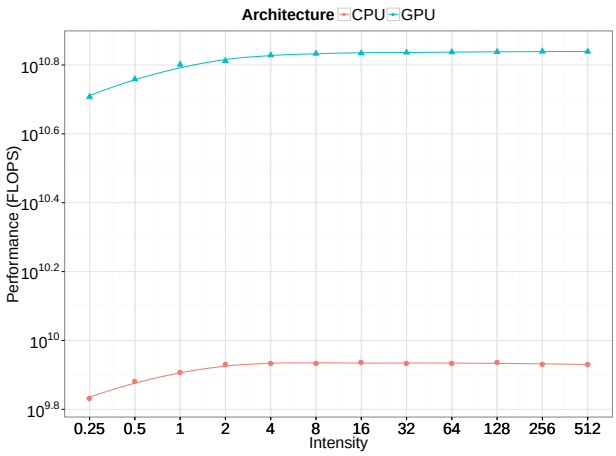

Figure 1: Performance roofline for the Snapdragon 805

point operations per second (flops) on the y-axis and various intensities on the x-axis. We obtain this roofline by running microbenchmarks that aim to maximize throughput for a given intensity [10]. The figure is consistent with previous work [3]; the GPU has a much higher upper-bound on performance than the CPU.

The data in Figure 1 is collected for several permutations of operational intensity (i.e., varying number of operations vs. bytes operated on). Choi et al. use this data in a regression model to estimate the cost of floating-point and memory operations [3]. The cost of an operation is separated into the time it takes for an operation to complete (τ) and the number of joules an operation consumes (ε). The cost is then multiplied by the number of operations and bytes used in an application to estimate the execution time and energy consumption.

An important caveat to this methodology is that it attempts to maximize throughput on the given architecture. The microbenchmarks used to obtain these rooflines utilize fused multiply-add (FMA) instructions that work on vectors [10]. However, many applications do not map well (or at all) to this type of microbenchmark. Unless your application can make heavy use of SIMD and FMA instructions, you are unlikely to ever reach the roofline. An alternative methodology is to segregate the data for each type of operation (e.g. add, multiply, etc.).[1] This methodology is used by Diop et al. [4], where multiple microbenchmarks are written that exclusively use one type of operation. Doing so provides coefficients for individual operations, which is applicable to different instruc-

[1] A discussion of control operations is deferred to Section 3.

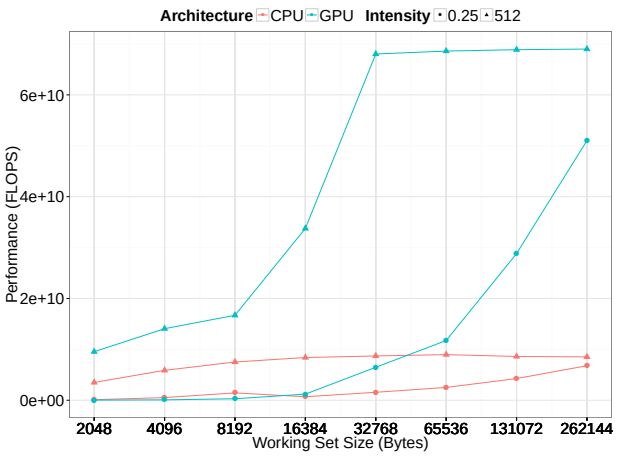

Figure 2: Performance bounds of the GPU and CPU for different working set sizes

tion mixes. But this still does not tell us which parts of an application are amenable to a GPU port. Identifying the regions of code that can definitively perform better on a GPU is a non-trivial task. We believe that combining the costs from CPU and GPU models with dynamic profiles of an application running on the CPU may be a good and objective indicator of what to send to the GPU.

3. Profiling the Application

Profiling applications running on the CPU is a quick way to understand the behaviour of an application. Figure 2 shows the performance that can be achieved for different working set sizes for two intensities using the FMA microbenchmark described in Section 2. The bottom lines for each architecture correspond to a low operational intensity (0.25), and the top lines to a high intensity (512). We can see that both architectures converge toward the roofline as we increase the number of bytes operated on. More importantly, however, we notice that intensity and working set size play a crucial role in the performance increase one can gain from porting to a GPU. At low intensities and low working set sizes (up to 16KB), the CPU and GPU provide comparable performance; it is only when we surpass 64 KB that the GPU begins to pull away at low operational intensities. Even at a high intensity, the GPU does not eclipse the CPU until 32KB (where it saturates at the roofline).

For the microbenchmarks under study, the behaviour differences between the CPU and GPU are mainly due to the working set data. For the platform studied, the data is already in the CPU's memory space. Avoiding transfer time allows the data to be more quickly loaded into the CPU's caches; we see little change in performance when changing the working set size (assuming it fits in the cache); our CPU has a 2 MB L2 cache. The GPU needs the data to be transferred to and from its own memory space. The GPU benefits from caching as well but the difference here is that the data is not available in the cache until we move the data explicitly to the GPU memory space. The memory transfer time for the GPU compared to the kernel's execution time has an impact on performance.

With smaller working sets and low intensities, performance is mainly influenced by the transfer time. Conversely, large working sets and high intensities mean that performance is influenced by the kernel's execution time. Therefore we can conclude that the amount of data operated on by the code and its operational intensity should factor into our decision on what to offload to the GPU.

Our assumption for this study was that a large portion of the input data set is available to the CPU in its caches. This is true for applications where the input is received from API calls (e.g., reading health data, data from a web server, file I/O) or from a previous code segment. As the CPU controls the flow of data between different sources, data must pass via the CPU (or CPU memory space) before the GPU can receive it. For example, if an application is monitoring the step count of a user over the past week, it must use native API calls to access this data, which subsequently becomes available to the CPU. The data can then be moved to the GPU's memory space. This model may change in the future if the operating system exposes the GPU to these API calls or devices directly or through mechanisms such as DMA.

Using heterogeneous system architectures [7] can also greatly improve the GPU's performance. These architectures provide a unified memory space for different devices (generally the CPU and GPU, but additionally other accelerators), removing the need to explicitly (or implicitly) copy data between different memory spaces. An example class of devices which exhibit this architecture are the AMD Fusion APUs [2]. In the future, it is likely that more architectures will follow this model. Our model can cater to this architecture as well. It eliminates the transfer time in our model if the data is already present in the unified memory space. We also eliminate the transfer time for estimates of applications where the GPU has direct access to the data (such as when reading images from the device camera.)

There are certain factors not illustrated by Figure 2. GPUs are simple cores (relative to CPUs). While they support the basic ISA (including flow control, arithmetic, logic, load/store operations), they typically do not have hardware optimizations such as branch predictors or data prefetchers [8]. Branch divergence also can also limit performance of control flow on GPUs. Even with a high operational intensity and large working set size, if the computation makes heavy use of branching it may not be a good fit for the GPU. One way to avoid (or delay) the analysis of branches is to analyse applications at the basic block granularity, which have only one point of entry and one point of exit. This way, we can quantify the impact of computations without control flow first.

Using a dynamic instrumentation framework such as Pin [9], we can instrument basic blocks to characterize their operational intensity, working set size, and instruction mix (each

operation has a count v_{op}). This is done using a serial version of the application developed for a generic CPU. During a run of the application, a basic block is executed a certain number of times (n). For each basic block, the total number of bytes, unique bytes accessed (reads and writes) and the number of useful operations performed are counted. Dividing the number of useful operations by the number of unique bytes *per execution* of the basic block gives us the intensity (ι). Counting the number of unique bytes accessed by the basic block over the *entire execution* of the application gives us its working set size (σ). Basic blocks with a low number of useful operations over the entire course of the application are likely to be serial code and can be left in the original CPU-side application. We are particularly interested in basic blocks with a large number of useful operations and/or a large number of iterations.

Once we have profiled the CPU application, we can multiply the characteristics by cost coefficients to estimate the run-time (T) and energy (E) of basic blocks on a CPU or GPU architecture. To do this, we construct microbenchmarks for arithmetic operations, memory operations, and memory transfer time. Coefficients for each of these are obtained via regression modelling (Section 2). Then we can objectively estimate the time a basic block will take to execute as:

$$T_{BBL} = max(\sum_{op} v_{op} \times \tau_{op}, \sigma \times \tau_{transfer}) \quad (1)$$

That is, the time a basic block takes to execute is the maximum of two calculations. The first calculation sums up the time taken for each operation executed by the basic block. The second calculation estimates the time to transfer the data to the GPU. We use the maximum because we assume the ideal case where transfer can be overlapped with computation [3]. However, Equation 1 does not capture intensity (ι), despite our findings in Figure 2. Unlike previous work, we believe that the regression coefficients should be modelled for each intensity, rather than collectively. The intensity of a basic block (or any portion of code) correlates to the amount of throughput one can achieve on a given architecture. Using one coefficient for all intensities will give time and energy estimates for low-intensity computations that are impossible to achieve in practice.

We can then estimate the total speed-up (or slowdown) and energy savings that we would achieve by offloading the computation performed by a basic block over the entire execution. This can be done on a per-basic-block level as well as relative to the entire application. Comparing the speed-up and energy saving gains to the entire applications allows us to observe which parts of an application are good candidates to be offloaded. We can then combine basic block sequences that frequently occur during a dynamic run of the program. The number of basic blocks in a sequence correlates to the number of branches needed; a penalty can be assigned based on the number of branches in this sequence.

4. Limitations

An important factor that our current methodology misses out on are data dependencies within and across basic blocks. Using our methodology may estimate speed-ups much higher than what can actually be achieved for such cases. An example of this would be generation of a cumulative probability density function for a given vector input.

Another class of applications that do not work particularly well with our methodology are those which require a different algorithm to have good performance gains when parallelized. One example of this is matrix multiplication. Using three nested *for* loops in the basic unoptimized form of the application does not allow multiple cores to exploit data locality during reads. To effectively parallelize this application, we need to heavily restructure the matrix multiply algorithm [5] to achieve good scalability and exploit memory locality. Our methodology would predict the speed-up based on the serialized version of the application, which likely would not correlate well with the parallelized algorithm. However, many modern applications are designed with parallelism in mind and are often multi-threaded for the CPU. This is particularly true when performance is an important factor such as for those developers seeking to accelerate their applications using the GPU.

5. Conclusion

In this position paper, we propose a methodology to estimate the benefit of offloading certain computations from a CPU to a GPU. The differences in performance for different intensities needs to be accounted for when deciding which computations work better on a CPU or GPU. It is unlikely that all basic blocks can achieve the maximum throughput an architecture has to offer. Therefore, a table of coefficients for various permutations of intensity and operations needs to be assembled (and memory transfer time). Combining these coefficients with basic-block profiling of an application will lead to a numerical, objective comparison of what can benefit from GPU acceleration. In the future, we plan to evaluate our methodology on a range of applications to determine its effectiveness in giving a fast, yet reasonable estimate prior to spending consider developer effort in porting the application.

References

[1] "Snapdragon 805 processors," https://www.qualcomm.com/products/snapdragon/processors/805, accessed: 2014-04-07.

[2] N. Brookwood, "AMD white paper: AMD fusion family of APUs," 2010.

[3] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 661–672.

[4] T. Diop, N. E. Jerger, and J. Anderson, "Power modeling for heterogeneous processors," in *Proceedings of Workshop on General Purpose Processing Using GPUs*. ACM, 2014, pp. 90:90–90:98.

[5] A. Gupta and V. Kumar, "Scalability of parallel algorithms for matrix multiplication," in *Parallel Processing, 1993. ICPP 1993. International Conference on*, vol. 3. IEEE, 1993, pp. 115–123.

[6] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and analysis of parallel block vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012, pp. 452–463.

[7] G. Kyriazis, "Heterogeneous system architecture: A technical review," *AMD Fusion Developer Summit*, 2012.

[8] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *Ieee Micro*, vol. 28, no. 2, pp. 39–55, 2008.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm Sigplan Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[10] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.