In Proc. *The International Conference on Engineering of Reconfigurable Systems and Algorithms* (ERSA'06), June 26 - 29, 2006, Las Vegas, Nevada

Code Partitioning for Reconfigurable High-Performance Computing: A Case Study

Volodymyr Kindratenko
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)
kindr@ncsa.uiuc.edu

Abstract

In this case study, various ways to partition a code between the microprocessor and FPGA are examined. Discrete image convolution operation with separable kernel is used as the case study problem and SRC-6 MAPstation is used as the test platform. The overall execution time of the resulting implementation serves as the primary optimization criterion. The paper presents an overview of the SRC-6 architecture and programming tools and describes the case study problem, along with a timing analysis of its microprocessor-based implementation. Next, three code partitioning schemes are considered and their SRC-6 MAP implementations are described, including detailed timing analyses. The results are compared and conclusions are drawn as to what partitioning scheme characteristics contribute most to the reduction of the overall execution time of the algorithm. The results of this case study are applicable to a large class of problems that involve outsourcing computationally demanding tasks to a reconfigurable processor.

1. Introduction

Reconfigurable computing (RC) [1] based on field programmable gate array (FPGA) technology has the potential to yield performance improvements beyond those predicted by Moore's Law [2]. Recently introduced commercial high-performance reconfigurable computing (HPRC) systems, such as Cray XD1, SGI RASC, and SRC-6 MAP™, which are based on the combination of conventional processors and FPGAs, enable software developers to exploit coarse-grain functional parallelism through conventional parallel processing as well as fine-grain parallelism through direct hardware execution on FPGAs. One of the key challenges in effectively using these systems is the need for manual partitioning of the algorithm between the microprocessor(s) and FPGA(s). How to partition the code such that the best overall application performance can be achieved is a fundamental research question. While some work has been done on automatic code partitioning [3-5], none of the obtained results have been implemented on the current production systems, such as SRC-6 MAP. It is up to the software developer to analyze the code and decide what should be ported to the FPGA and what should be left on the microprocessor.

Some well-understood common metrics, such as the number of operations and results per data unit, data reuse efficiency, data per latency, etc. [6], can be useful to guide the partitioning process. Yet there are other practical considerations, such as the number of times the FPGA function is called, the number of times the direct memory access (DMA) engine is invoked, and microprocessor data manipulation tasks, that may have an adverse effect on the overall algorithm performance. The goal of this case study is to examine what impact different code partitioning schemes, which are similar in the common metrics space [6] but differ in other ways, have on the overall algorithm performance.

The case study is based on an example of an image convolution algorithm that uses a separable convolution kernel. This particular algorithm enables us to consider three levels of code partitioning granularity. At the lowest level, only the core computational kernel is outsourced to the FPGA and the microprocessor is left to deal with the memory manipulation tasks. At the intermediate level, the algorithm is partitioned along the lines of two major computational tasks. And at the highest level, the entire algorithm is ported to the FPGA. These code partitioning schemes demonstrate the impact of different levels of partitioning granularity on the overall code performance and the FPGA code

complexity. The observations made in the paper are intended to serve as the guidelines that one might refer to when considering porting code to an RC platform.

SRC-6 MAP [7] was used in this case study as the target platform because it is one the most readily available production RC systems on the market. The development toolset, called Carte [8], also provides a clear path for code development on the FPGA as well as a convenient debugging and simulation environment.

2. Case study problem

The MATPHOT code [9] used in stellar photometry and astrometry is the application driver for this work. The core of the code is a discrete convolution operation that convolves a synthetic image with a 21 coefficient-wide damped sinc function, $\mathrm{sinc}(x)=\sin(\pi x)/\pi x$, using a separable kernel. In MATPHOT, single precision floating point numerical resolution is required for both the synthetic image and damped sinc function.

The basic idea of image convolution is that a window of some finite size and shape, $h[k,l]$, is scanned across the image and the output pixel value is computed as the weighted sum of the input pixels, $a[m,n]$, where the weights are the values of the filter assigned to every pixel of the window:

$$a[m,n] \otimes h[k,l] = \sum_{i=0}^{k-1} \sum_{j=0}^{l-1} a[m+i, n+j] h[i,j]$$

The window with its weights is called the *convolution kernel*. The *per-pixel* computational complexity for a KxL convolution kernel is O(KL).

If the convolution kernel $h[k,l]$ is *separable*, that is, if the kernel can be written as

$$h[k,l] = h_{row}[l] \bullet h_{col}[k]$$

then the convolution can be performed as follows:

$$a[m,n] \otimes h[k,l] = \sum_{i=0}^{k-1} \left\{ \sum_{j=0}^{l-1} a[m+i, n+j] h_{row}[j] \right\} h_{col}[i]$$

Thus, instead of applying one two-dimensional convolution kernel, it is possible to apply two one-dimensional kernels: the first one in the l direction and the second one in the k direction. This reduces the per-pixel computational complexity to O(K+L).

Microcomputer implementation of the last equation is straightforward: For an MxN image, $a[m,n]$, a one-dimensional convolution with $h_{\mathrm{row}}[l]$ kernel is performed for each row of pixels followed by a convolution with $h_{\mathrm{col}}[k]$ kernel for each column:

```
2DCONVOLUTION(A, B, M, N, Hr, Hc, L, K)
1   for m ← 0 to M-1
2       for n ← 0 to N-1
3           R1[n] ← A[m, n]
4       R2 ← 1DCONVOLUTION(R1, N, Hr, L)
5       for n ← 0 to N-1
6           B[m, n] ← R2[n]
7   end
8   for n ← 0 to N-1
9       for m ← 0 to M-1
10          C1[m] ← B[m, n]
11      C2 ← 1DCONVOLUTION(C1, M, Hc, K)
12      for m ← 0 to M-1
13          B[m, n] ← C2[m]
14  end
15  return B
```

Here **A** denotes input image, **B** denotes output image, both of dimension MxN, **Hr** denotes the convolution kernel (consisting of L elements) applied to each row, and **Hc** denotes the convolution kernel (consisting of K elements) applied to each column. Lines 1-7 correspond to per-row convolution: Pixels from each row are copied to a separate array (lines 2-3), **R1**, a one-dimensional convolution with the appropriate coefficients is performed on **R1** (line 4), and the results are copied to the destination image **B** (lines 5-6), which is then processed in a similar manner for each column (lines 8-14). Finally, the following is the 1DCONVOLUTION subroutine:

```
1DCONVOLUTION(I, O, P, H, Q)
1   for p ← 0 to P-1
2       O[p] ← 0
3       for q ← 0 to Q-1
4           O[p] ← O[p] + I[p+q] · H[q]
5   end
6   return O
```

The 2DCONVOLUTION algorithm is the subject of the present study. Its computational complexity is O((K+L)MN); thus, for a fixed-size convolution kernel the overall execution time of the algorithm is the function of image size.

3. Case study platform

The SRC-6 MAPstation [7] used in the course of this study consists of a commodity dual-CPU Xeon

board, a MAP Series C processor, and an 8 GB common memory module, all interconnected with a 1.4 GB/s low-latency switch. The SNAP™ Series B interface board is used to connect the CPU board to the Hi-Bar switch. The SNAP plugs directly into the mother board's DIMM memory slot.

The MAP Series C processor module contains two user FPGAs, one control FPGA, and memory. There are six banks (A-F) of on-board memory (OBM); each bank is 64 bits wide and 4 MB deep for a total of 24 MB. The programmer is responsible for data transfer to and from these memory banks via SRC programming macros invoked from the FPGA application. There is an additional 4 MB of dual-ported memory dedicated solely to data transfer between the two FPGAs.

The two user FPGAs in the MAP Series C are Xilinx Virtex-II XC2V6000 FPGAs. Each FPGA contains 6 million equivalent logic gates, 144 dedicated 18x18 integer multipliers, and 324 KB of internal dual-ported block RAM (BRAM). These FPGA elements are not directly visible to the programmer but are interconnected appropriately as determined by the programmer's MAP C algorithm code, the SRC Carte programming environment tools, and the Xilinx FPGA place and route tools. The FPGA clock rate of 100 MHz is set by the SRC programming environment.

The Carte programming environment [8] for the SRC MAPstation is highly integrated, and all compilation targets are generated via a single makefile. The two main targets of the makefile are a debug version of the entire program and the combined microprocessor code and FPGA hardware programming files. The debug version is useful for code testing before the final time-intensive hardware place and route step. Either the Intel icc compiler or the gcc compiler can be used to generate both the CPU-only debug executable and the CPU-side of the combined CPU/MAP executable. The SRC MAP compiler is invoked by the makefile to produce the hardware description of the FPGA design for final combined CPU/MAP target executable. This intermediate hardware description of the FPGA design is passed to the Xilinx ISE place and route tools, which produces the FPGA bit file. Lastly, the linker is invoked to combine the CPU code and the FPGA hardware bit file(s) into a unified executable.

4. Code partitioning alternatives

The core of the computation is a fixed-width 21-coefficient 1D convolution operation that uses single precision floating point arithmetic. XC2V6000 FPGA has enough hardware multipliers to implement just two such 21-coefficient-wide fully unrolled operations (42 single precision floating point multiplications and 40 additions in total) using SRC's reduced space multiply macros. Therefore, our ability to perform simultaneous convolution operations is limited by the available FPGA hardware resources to just two such operations.

Considering the *overall execution time* of the algorithm as the main efficiency criteria, and taking into account availability of FPGA resources, what is the most efficient way to partition the 2DCONVOLUTION algorithm between the microprocessor and MAP processor?

To answer this question, we examine several partitioning options and investigate their run-time behavior. Perhaps the simplest partitioning approach is to outsource the 1DCONVOLUTION algorithm alone. Alternatively, the entire convolution operation in one dimension can be implemented on MAP. And finally, the entire 2DCONVOLUTION algorithm can be ported to MAP. Note that these partitioning alternatives result in the same number of calculations to be performed on MAP, thus, they are similar in the sense of the common metrics used in [6] (with the exception of data reuse efficiency), yet they are very different as far as the number of times the MAP subroutine is called and the type and amount of memory manipulations with which the microprocessor is left.

4.1. Code partitioning choice # 1

It is natural to consider a partitioning scheme in which the 1DCONVOLUTION subroutine alone is outsourced to the MAP processor. An obvious advantage of this approach is its simplicity: One is concerned with only one row or column of image pixels at a time without explicitly distinguishing between them, which simplifies the data management aspects of MAP code implementation. The main disadvantage of this approach, of course, is the need to call the MAP-based 1DCONVOLUTION function multiple times, thus likely encountering some MAP function call overhead that may have an effect on the overall performance.

Porting 1DCONVOLUTION to MAP is straightforward. There is enough space and hardware multipliers on the MAP's primary FPGA chip to perform two sets of convolution calculations in parallel using SRC's floating point single precision smaller area macros. Therefore, the overall MAP code sequence deployed on just one chip is:

- If run the first time, DMA from main RAM convolution coefficients to OBM bank F
- Copy convolution coefficients from OBM bank F to on-chip registers
- DMA from the RAM pixel values to OBM bank A
- Do calculations using pixels from OBM bank A and storing results in OBM bank B
  - Bring in the next two pixel values from OBM bank A to the on-chip pixel registers
  - Shift the on-chip pixel registers by two pixels
  - Perform two parallel convolution calculations
  - Store two results in OBM bank B
- DMA to main RAM results from OBM bank B

The mapped and routed FPGA implementation of the 1DCONVOLUTION subroutine occupies all available SLICEs and 91% of all available MULT18X18s on one MAP Series C processor's FPGA and meets timing requirements of 9.998 ns.

4.2. Code partitioning choice # 2

The next partitioning scheme is based on the observation that in the CPU implementation the entire image is located in a continuous memory array, one row of pixels after another. Therefore, it is straightforward to have access to the consecutive rows of image pixels without copying them to a separate array. Thus, the per-row convolution calculations for the entire image can be outsourced to MAP, literally by replacing lines 1-7 in the 2DCONVOLUTION algorithm with just one call to a MAP-based subroutine. Once all rows of the image are processed, the image data must be rearranged in the memory so that the columns occupy a continuous memory array, one column of pixels after another. Then the same MAP subroutine can be called on the rearranged image with the net effect of performing per-column convolution calculations. At the end, the pixels are moved back to their original locations. The overall MAP code sequence for this implementation is:

- DMA in convolution coefficients to OBM bank F
- Copy convolution coefficients from OBM bank F to on-chip registers
- For each row of image pixels
  - DMA in pixel values to OBM bank A
  - Do calculations using pixels from OBM bank A and storing results in OBM bank B
  - DMA out results from OBM bank B

Conceptually, the MAP subroutine of this implementation is very similar to the previous implementation; we just augmented the previously written code with an extra loop responsible for bringing in and out the next row/column of data rather than leaving this to the microprocessor. The microprocessor code now is left with the extra work needed to rearrange the image data in memory.

This implementation occupies all available SLICEs, some of which are packed with unrelated logic, and 95% of all available MULT18X18s on one MAP Series C processor's FPGA and meets timing requirements of 9.994 ns.

4.3. Code partitioning choice # 3

In this partitioning scheme, the entire 2DCONVOLUTION algorithm is ported to MAP. However, there are some difficulties with implementing this approach in practice. Note that the previous design occupied all available SLICEs on the MAP Series C processor's FPGA, some of which were already packed with unrelated logic. Therefore, the primary FPGA is used to implement the calculations in one image dimension and the secondary FPGA is used to implement the calculations in the other image dimension. The intermediate image is stored in the on-board memory, which limits the size of the image that can be processed by this implementation to 12 MB. The primary FPGA MAP code sequence is:

- DMA in one set of convolution coefficients to OBM bank E
- DMA in the other set of convolution coefficients to OBM bank F
- Copy convolution coefficients from OBM bank F to on-chip registers
- DMA in input image to OBM banks A-C
- For each row of image pixels
  - Do calculations using pixel values from OBM banks A, B, and C and storing results in OBM banks D, E, and F
- Let other chip to do per-column calculations
- DMA out results from OBM banks A-C

The FPGA chips on the MAP processor operate in a master-slave mode. The secondary chip waits until the primary chip is done with the per-row calculations and only then performs per-column calculations. The secondary FPGA MAP code sequence is:

- Copy convolution coefficients from OBM bank E to on-chip registers
- Wait until the primary chip is done with per-row calculations

- For each column of image pixels
  - Do calculations using pixel values from OBM banks D, E, and F and storing results in OBM banks A, B, and C

This implementation occupies all available SLICEs and over 95% of MULT18X18s on both chips and meets timing requirements of 9.995 ns. However, it took some effort to fine-tune the secondary FPGA design to fit on the chip and meet timing requirements.

5. Implementation results and discussion

Let us first examine the original microprocessor-only implementation described in Section 2. Figure 1 shows how the overall execution time changes as the image size increases. It also shows what fraction of time is spent due to memory copy operations (lines 2-3, 5-6, 9-10, 12-13 of the 2DCONVOLUTION algorithm) and due to the actual calculations (lines 4 and 11). Thus, for a 2,048x2,048 pixel image, memory copy operations are responsible for about 0.72 seconds, whereas actual convolution calculations take about 0.62 seconds. (Calculations reported in this paper were performed on a 2.8 GHz Intel dual-Xeon platform; code was compiled with the gcc 3.4.3 compiler using the O3 optimization level. The microprocessor "Read Time Stamp Counter" instruction (RDTSC) [10] was used to measure timing information.)

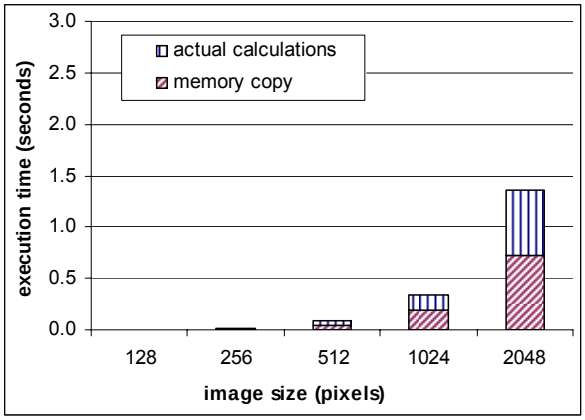

Figure 1. Time to compute vs. image size for 2DCONVOLUTION algorithm. The horizontal axis shows image dimensions, thus "512" means an image consisting of 512x512 pixels.

Figure 2 provides test results for the first partitioning scheme implemented as described in Section 4.1. As with the microprocessor-only implementation, a significant amount of time is spent due to the memory copy operations, whereas time spent performing actual calculations is only marginally smaller than in the native CPU-only implementation. Note that 'MAP code execution' time includes both data transfer and convolution calculations time. However, an even larger amount of time is now spent due to the MAP function call overhead. We measure this overhead as the difference between the time spent on the CPU while executing the MAP function and the time measured inside the MAP function while executing its internals (including data transfer) on the FPGA. In other words, MAP function call overhead is what it takes to call an "empty" MAP function that returns immediately without any work done.

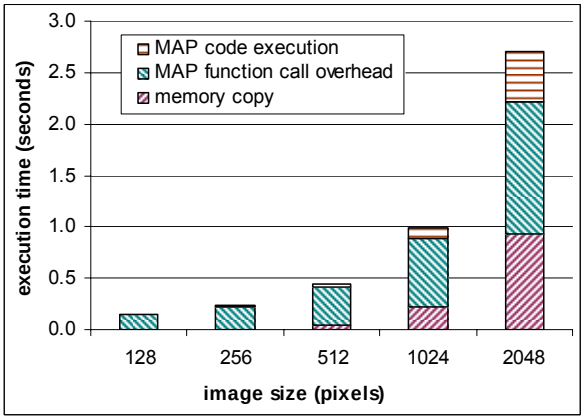

Figure 2. Execution time for 2DCONVOLUTION algorithm in which 1DCONVOLUTION subroutine alone is ported to MAP.

Figure 3 provides test results for the second partitioning scheme implementation described in Section 4.2. The MAP function call overhead, which was a major issue with the previous code partitioning scheme, is now independent of the image size (since the MAP function is called only twice) and became much smaller. The MAP code execution time increased as compared to the previous implementation. However, the amount of time spent due to the memory copy operations on the CPU remains about the same as with our previous implementation, even though each pixel value is copied only twice, whereas in the previous implementation it was copied four times, although in smaller memory segments. This is likely due to the CPU memory cache misses.

Note that the actual calculation time of the MAP implementation can still be reduced if we involve the second FPGA chip available on the MAP Series C processor. But even this will not reduce the overall algorithm execution time with any significance since the time spent due to the image rearrangement on the

CPU accounts for the majority of the execution time. The need to "rotate" the image twice in the system memory resulted in a significant time overhead.

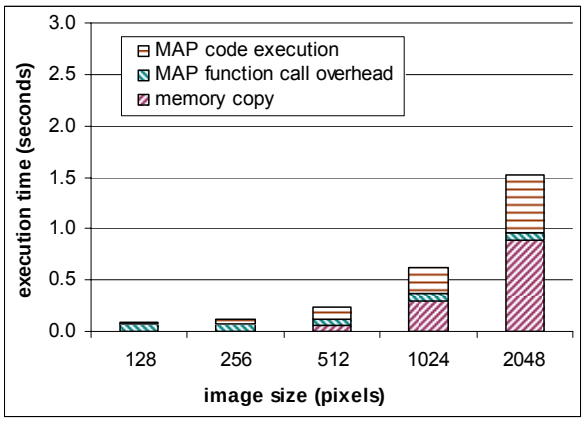

Figure 3. Execution time for 2DCONVOLUTION algorithm in which the entire convolution operation in one image dimension is outsourced to MAP.

Figure 4 provides test results for the third partitioning scheme implemented as described in Section 4.3. In this implementation, all the microprocessor-side calculations and memory manipulations have been eliminated. The MAP subroutine is called only once, therefore the MAP function call overhead remains small and independent of the image size. This overhead, however, is doubled as compared to the previous implementation due to the fact that now both FPGA chips are used. On the other hand, the actual calculation time measured on the MAP decreased significantly since the DMA engine is invoked from the FPGA design only twice, once to transfer in the entire image and once to transfer out. (Remember that in the previous implementation the DMA engine was invoked twice per each image row/column.) Thus, for an image consisting of 1,772x1,772 pixels, we achieved a 3x overall code execution speedup.

Figure 5 shows a combined comparison chart of the execution time for all four implementations. It is clear that the first code partitioning scheme suffers due to the MAP function call overhead. Even though this partitioning scheme is intuitive and simple to implement, it increases the overall execution time because the MAP function is called frequently and thus the accumulated MAP function call overhead adds up quickly to the overall execution time.

The second code partitioning scheme suffers due to the need to perform costly memory manipulations on the microprocessor and also due to the need to invoke the DMA data transfer engine multiple times.

The third code partitioning scheme eliminates the need for any memory manipulations on the microprocessor side. It also eliminates the need for the frequent use of the DMA data transfer engine as the entire image data is transferred in and out only once. As a result, the MAP code execution time is very short and the overall execution time is dominated by the MAP function call overhead.

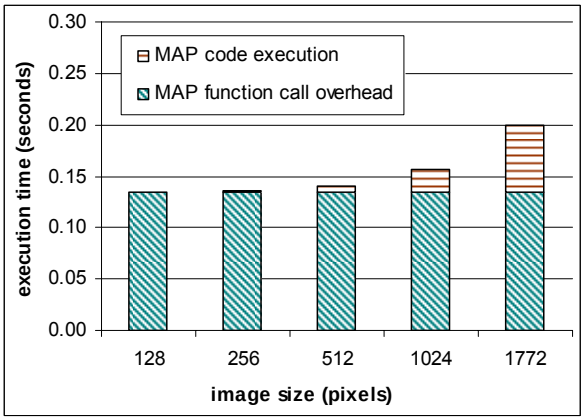

Figure 4. Execution time as a function of image size for 2DCONVOLUTION algorithm implemented solely on MAP. Note the vertical axis scale difference between this figure and Figures 1- 3. Also note that the largest image that can be processed by this implementation is 1,772x1,772 pixels.

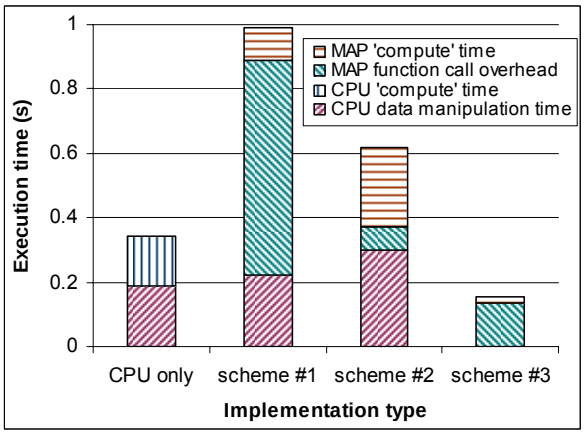

Figure 5. Execution time comparison chart for a 1,024x1,024 pixels image.

The 2DCONVOLUTION algorithm is an interesting case to study since the best way to partition it is not immediately clear based on the run-time analysis of the algorithm shown in Figure 1. It is tempting to port the 1DCONVOLUTION algorithm alone since it is responsible for about half of the overall execution time. Yet this resulted in a

significant MAP function call overhead that increased the overall execution time (Figure 2). Porting the entire 2DCONVOLUTION algorithm presents some challenges since it requires both MAP FPGAs to be utilized at their full capacity, thus making it difficult to meet timing requirements. Yet this approach yields the best overall performance.

6. Conclusions and future work

All three partitioning schemes presented in this paper resulted in the same number of calculations executed on the FPGA when combined across a single run of the 2DCONVOLUTION algorithm. Yet they resulted in very different execution times. This points out the importance of the overall code organization for reconfigurable system applications. The MAP function should be called as few times as possible in order to eliminate the MAP function call overhead. A partitioning scheme that reduces or eliminates the need for data manipulations by the microprocessor should be considered. The DMA engine should be invoked in the MAP code as few times as possible since it adds considerable overhead to the MAP code execution. Thus, when considering different code partitioning alternatives, in addition to the metrics based on various aspects of data reuse such as those reported in [6], one should also take into account other practical considerations, such as the number of times the MAP code will be invoked, amount if extra memory manipulation tasks left to the microprocessor, etc.

We have not seen MAP function call overhead timing measurements reported in the literature and our own estimates vary. For example, combined MAP function call overhead for the design described in Section 4.2 is 0.068 seconds, whereas the overhead for the design provided in Section 4.3 is 0.135 seconds. In the first case, only one FPGA chip was used and the MAP function was called twice. In the second case, both FPGAs were used, but the MAP subroutine was called only once. Our estimates show that the first time a MAP subroutine is called it encounters a 67 millisecond overhead due to the need to load the FPGA configuration bitfile. Each consecutive call to the same MAP function resulted in an overhead that varied for different designs. The nature of this variability and the ways it can be precisely measured and/or predicted is the subject of the future work.

7. Acknowledgements

This work was funded by the National Science Foundation (NSF) grant SCI 05-25308. MATPHOT software was developed and kindly provided to us by Dr. Kenneth Mighell from the National Optical Astronomy Observatory. Special thanks to David Caliga, Dan Poznanovic, and Jeff Hammes, all from SRC Computers Inc., for their help and support with SRC-6 system. Further comments and suggestions were provided by David Pointer, Dr. Craig Steffen and David Raila from NCSA's Innovative Systems Laboratory. Special thanks to Trish Barker from NCSA's Office of Public Affairs for help in preparing this publication.

7. References

[1] M.B. Gokhale, and P.S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*, Springer, Dordrecht, 2005.

[2] K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating Point Performance," In Proc. *12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 2004, pp 171-180.

[3] T. Callahan, J. Hauser, and J. Wawrzynek, "The GARP architecture and C compiler". *IEEE Computers*, 33:4 (April 2000), pp. 62-69.

[4] Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood, "Hardware-software co-design of embedded reconfigurable architectures", In Proc. *Design Automation Conf.*, 2000, pp. 507--512.

[5] B. Knerr, M. Holzer, and M. Rupp, "HW/SW Partitioning Using High Level Metrics", In Proc. *Int. Conf. on Computing, Communications and Control Technologies*, 2004, pp. 33-38.

[6] V. Kindratenko, D. Pointer, and D. Caliga, "High-Performance Reconfigurable Computing Application Programming in C", *White Paper*, January 2006. netfiles.uiuc.edu/dpointer/www/whitepapers/hprc_v1_0.pdf

[7] SRC Computers Inc., Colorado Springs, CO, *SRC Systems and Servers Datasheet*, 2005.

[8] SRC Computers Inc., Colorado Springs, CO, *SRC C Programming Environment v 1.9 Guide*, 2005.

[9] Mighell, K. J., "Stellar photometry and astrometry with discrete point spread functions", *Monthly Notices of the Royal Astronomical Society*, 316, 861-878 (11 August 2005).

[10] *IA-32 Intel® Architecture Software Developer's Manual*, Volume 3B: System Programming Guide, Part 2.