High-Performance Reconfigurable Computing Application Programming in C

Volodymyr Kindratenko and David Pointer‡
National Center for Supercomputing Applications (NCSA)
University of Illinois at Urbana-Champaign (UIUC)
1205 W. Clark St. Room 1008
Urbana, IL 61801
{kindr, pointer}@ncsa.uiuc.edu

David Caliga
SRC Computers, Inc.
4240 N. Nevada Avenue
Colorado Springs, CO 80907
caliga@srccomp.com

1 Introduction

At NCSA, scientists aggressively seek floating point application performance improvements far beyond those implied by Moore's Law. The Innovative Systems Lab at NCSA was created to explore emerging computer technologies that have the potential to serve the application scientists' mid- to long-term performance requirements. These technologies include cell processing, specialized ASIC application coprocessors, multicore processors, graphics accelerators used for general purpose applications, and reconfigurable computing (RC), that is, computers that utilize devices such as Field Programmable Gate Arrays (FPGA) as processor elements. In this white paper, we focus on reconfigurable computing application programming.

SRC Computers, Inc. is an early pioneer in the field of reconfigurable computing. SRC's work is based on the belief that in the near future, primarily because of processing costs and requisite volumes, only high end microprocessors and reconfigurable devices will be built with leading edge process geometries. The classic von Neumann load/store architecture is very good for some tasks, but seems to have serious scalability issues. On the other hand, reconfigurable devices have proven to be very scalable for a large class of applications, as well as having technology performance increase curves that far exceed Moore's law[1]. It is this realization that led SRC to develop its IMPLICIT+EXPLICIT™ Architecture[2]. SRC has been developing systems based on this architecture since 1997. This architecture combines both traditional

‡ Corresponding author
[1] K. Underwood, FPGAs vs. CPUs: Trends in Peak Floating Point Performance, *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, Monterey (CA), February 22-24, 2004, pp 171-180.
[2] IMPLICIT+EXPLICIT Architecture White Paper available by request at http://www.srccomp.com/WhitePapers.htm

microprocessors and reconfigurable processors, called MAP®, as peers on a shared memory, and has yielded orders of magnitude performance increases for many applications.

Since shipping their first system in 1998, SRC has continued to enhance the overall system hardware implementation, as well as developing a complete high level language programming environment called Carte™. Carte eliminates the need for the programmers to have an in-depth hardware design knowledge by allowing them to use standard ANSI C and FORTRAN, as well as standard programming constructs, to program both the standard microprocessors and the MAP processors. Carte also includes a complete debugging environment to facilitate program development in way that is very familiar to programmers.

For the past 15 years, FPGA designs were implemented by hardware engineers using a specialized hardware description language (HDL), like Verilog or VHDL. This is appropriate for embedded applications, but with recent improvements in FPGA technology there is great potential for reconfigurable computing to provide scientific application performance beyond Moore's Law. Industry has responded by developing several C and C-like languages that target FPGAs instead of CPUs. After all, application scientists should not be expected to become hardware engineers in order to utilize a new computer technology.

At the same time, application programmers and scientists who target reconfigurable systems need to be aware that a C compiler on these machines is not "just another C compiler". New system architectures require that the programmer learn a new set of best practices and rules of thumb ("application programming") that work well for the new architecture. Put another way, the application programming style that works well for scalar CPU programming will not map well to a combined CPU/FPGA reconfigurable architecture. At NCSA, we are working closely with application scientists and industry application specialists to develop effective programming styles and best practices on reconfigurable systems. This paper introduces reconfigurable system architecture and describes our approach to reconfigurable system application programming.

1.1 Why Reconfigurable Computing?

Learning a new architecture and a new programming style is not a trivial task, and so a reasonable question might be why bother with reconfigurable computing in the first place? The simple answer is that reconfigurable systems offer potential application performance improvements beyond those predicted by Moore's Law. The potential lies in the nature of the architecture: an application programmer may develop a dedicated, special purpose hardware compute engine that exploits the concurrency inherent in their application. This is exactly what drives the Application Specific Integrated Circuit (ASIC) development of devices such as Grape[1] and Clearspeed[2]. But while ASICs take months to develop, have a substantial tooling cost, and can not be changed, FPGA based designs take days to weeks to develop, have no tooling cost,

[1] T. Kuberka, A. Kugel, R. Männer, H. Singpiel, R. Spurzem, and Ralf Klessen, AHA-GRAPE: Adaptive Hydrodynamic Architecture – GRAvity PipE, *Proceedings of The 1999 International Conference on Parallel and Distributed Processing Techniques and Applications* (PDPTA 99), Las Vegas (NV) USA June 28-July 1, 1999, pp 1189-1195.

[2] Clearspeed Technology plc, CSX Architecture Whitepaper, PN-1105-0003, San Jose (CA) USA, 2005.

and can easily be changed. Even better, ASIC design requires extensive hardware engineering expertise not required for reconfigurable system application development.

1.2 Concurrency

We state in section 1.1 that an FPGA may be used to exploit the concurrency inherent in an application's algorithm. What do we mean by concurrency? There are two types of concurrency that may be inherent in an algorithm: *deep parallelism* and *wide parallelism*.

Deep parallelism (Figure 1), also known as pipelining or instruction level parallelism, is used when a function's set of operations is broken up into stages, where each stage executes in a single clock cycle and each stage's input depends solely on the results of the previous stage. The latency of the pipeline is the number of clock cycles needed to get the first result from this set of pipelined operations and is equal to the number of pipeline stages. The good news is that as long as the pipeline has input data available every single clock cycle, a result is generated every single clock cycle after the latency penalty is paid. If there is a lot of data relative to the pipeline latency, then the time to get the first result appears insignificant. Put another way, the processing time of a well-designed pipeline is O(*N*+*L*), where *N* is the size of the data to be processed and *L* is the number of stages in the processing pipeline. If *N* is large relative to *L*, then the processing time approaches O(*N*).

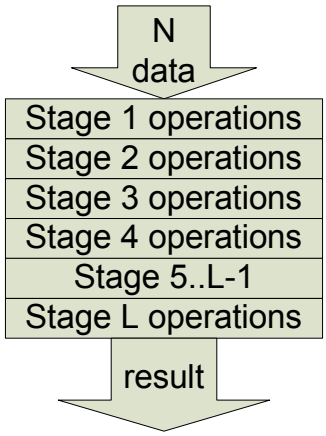

Figure 1 - Deep Parallelism

It is good to note that the SRC compiler technology automatically pipelines a set of operations inside a loop unless directed otherwise by the programmer.

Wide parallelism (Figure 2), also known as data level parallelism, simply duplicates a single pipeline and splits the input data across the duplicated pipelines. This is true parallelism, as the pipelines are actually implemented as separate entities in FPGA hardware and execute simultaneously. This, of course, only works when the input data consists of independent elements. The execution time of this type of parallelism is O((*N*+*L*)/*P*) where *N* is the size of the data to be processed, *L* is the number of stages in the processing pipeline and *P* is the number of independent pipelines. The ideal number of independent pipelines varies from algorithm to algorithm, but in general, we run out of FPGA resources before we reach an ideal pipeline count.

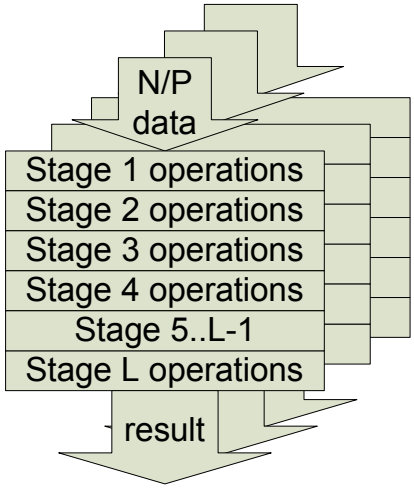

Figure 2 - Wide Parallelism

This level of algorithm parallelism needs to be explicitly implemented by the programmer and requires the use of SRC compiler directives.

1.3 CPU vs. FPGA Clock Speeds

There is an interesting question relating current FPGA clock speeds and CPU clock speeds. At present, a typical CPU runs at 3 GHz, while a typical FPGA runs at 100 MHz (Figure 3). This speed mismatch appears crippling to the FPGA, since the CPU can execute 30 one cycle serial operations in the same time that it takes the FPGA to execute just 1 one cycle serial operation. But consider this: a CISC CPU takes many one cycle clocks to execute a single instruction. The CPU also needs to service interrupts, executes time slices for all of the system processes, and has memory latency and register spill effects. On the other hand, the FPGA is a dedicated algorithm specific coprocessor that does nothing but generate results for an application. To date at NCSA, we have found that for large real-world data sets with many pipelined single precision floating point operations, combined CPU/FPGA execution may achieve from 3x to 40x performance improvement over the CPU execution alone.

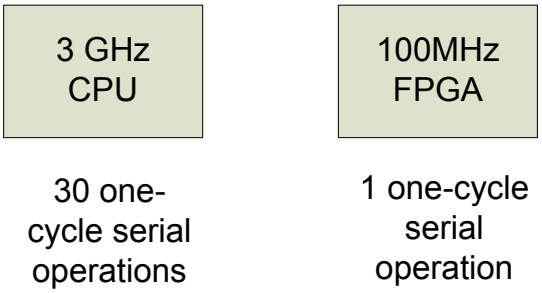

Figure 3 - CPU vs. FPGA Clock Speeds

1.4 How to get started with Reconfigurable Computing?

Simply stated, the first step is to find a core algorithm in an application and completely understand the required data movement into and out of the algorithm.

A scientific application code typically has at least one computationally intensive algorithm embedded in it. Presumably, this algorithm is where the application spends most of its execution time. Perhaps the algorithm calculates forces between atoms or calculates the distance between background and foreground pixels or performs string matching. Regardless, an algorithm may be abstracted from the application so that it appears in the algorithm form shown in Figure 4: a result or a set of results is obtained by a series of operations on the function input operands.

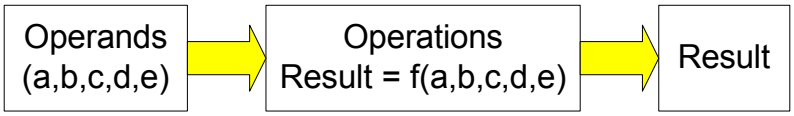

Figure 4 - Abstract Algorithm Model

Once the programmer has a good understanding of the operand data movement into the algorithm, the algorithm operations on the operand data, and the movement of the resulting data, they may begin to map their algorithm to reconfigurable system architecture.

For instance, consider the 2D image distance transform algorithm[1] used in many image processing applications in which image regions need to be isolated for subsequent processing. The algorithm is simply stated: for each background pixel, calculate the distance to the nearest foreground pixel. Thus, the resulting pixel value for each background pixel is the distance from that background pixel to the nearest foreground pixel.

So, in its simplest form, the distance transform algorithm calculates the distance to all pixels in the foreground for a single given background pixel. This simple form is repeated for each background pixel. Described programmatically,

```
int m,n
long bg_x[BG_MAX], bg_y[BG_MAX]
long fg_x[FG_MAX], fg_y[FG_MAX]
float d, distance[BG_MAX]

for m = 0 to BG_MAX-1
        for n = 0 to FG_MAX-1
                d = calculate_distance(bg_x[m], bg_y[m], fg_x[n], fg_y[n])
                if(d < distance[m])
                        distance[m] = d
```

The input operands to the algorithm are the two lists of pixels: background pixel coordinates and foreground pixel coordinates. A result is obtained for a background pixel *m* after the distance to all foreground pixels from pixel *m* have been calculated. The operations in the "calculate_distance" function are from the Euclidian distance equation

[1] P. E. Danielsson, "Euclidean distance mapping," *Comput. Graphics Image Processing*, vol. 14, pp. 227-248, 1980.

$$d = \sqrt{(bg\_x - fg\_x)^2 + (bg\_y - fg\_y)^2} \qquad (1)$$

At this point, we understand the algorithm's operand data input movement, the algorithm's operations, and the resulting data output movement. We will continue to use this algorithm as a reconfigurable system design example throughout this paper.

2 Reconfigurable System Architecture

A simple view of reconfigurable system architecture is shown in Figure 5. One or more FPGAs and dedicated FPGA memories are connected to CPU main memory via some high-bandwidth interface. Arrays of data from the CPU memory are transferred over this interface to the FPGA memory and streamed through the computation engine(s) inside the FPGA. Results of this computation are transferred to FPGA memory and streamed back to CPU memory.

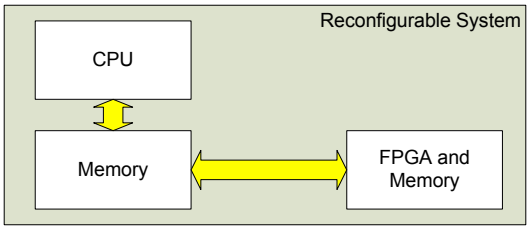

Figure 5 - Simple Reconfigurable System Architecture

2.1 What is an FPGA?

A computer hardware designer has a very detailed and specific view of the inside of FPGAs. However, our focus is on FPGA application programming with high level languages, so we will omit a great deal of detail and focus only on the aspects of FPGAs important to a scientific application developer.

An FPGA is an integrated circuit in some sense very similar to a CPU. A CPU is an integrated circuit that contains many functional units used to interpret and execute program instructions. An FPGA, on the other hand, is essentially a collection of *unconnected* simple and complex functional units in hardware – the application programmer's code defines the connections among the functional units. Some of these hardware functional units are very complex and specific – 18 bit integer multipliers, for example. Most of the functional units are no more than simple hardware building blocks that may be interconnected to make more complex functional units.

The idea of CPU instruction execution does not exist in FPGAs. Rather, the functional blocks are defined and interconnected by the development process to provide the hardware units used to directly execute the calculations and operations specified by the programmer. Unlike a CPU, all of the hardware functional units are specified by the programmer – the FPGA is completely dedicated to solving the programmer's algorithm. Even better, all of the hardware functional units in a programmed FPGA may execute in parallel.

Before moving on, let's quickly outline the development process for FPGAs in reconfigurable systems. An algorithm's operations and data movement are specified in C by the programmer.

This embodiment of the algorithm is compiled to some intermediate hardware description language and given to the FPGA vendor's tools. These tools perform the functional unit placement in the FPGA and determine the functional units interconnect ("place and route"). After place and route, which can take some time, the FPGA vendor tools produce an FPGA configuration file, or a "bit file". This bit file may be loaded onto the FPGA ("programming the FPGA") to set the functional unit interconnect in the target device. After programming the FPGA is complete, is available for use until the system is powered down or a different bit file is programmed into the FPGA.

The programmer need only be concerned about the very first step – using C to specify an algorithm's operations and data movement. The intermediate steps are usually hidden from the programmer and handled by the development tools, although this may vary from vendor to vendor.

2.2 FPGA and Memory

FPGA local memory is typically organized into several banks attached to the FPGA. In the SRC MAPstation MAP® Series C module, there are six banks of 64 bit wide memory around two FPGAs. (Either FPGA may access any memory using the compiler's FPGA synchronization primitives.) For this discussion, we will use all six banks and one FPGA. Five banks will contain operands and one bank will contain results. The FPGA calculates each element of the result array from some function of each element of the five input arrays, as shown in Figure 6. The full operation of the algorithm starts with the transfer of arrays 0 through 4 from allocated areas of CPU memory to five banks of the FPGA memory. Data from these five banks stream simultaneously into the FPGA and the function *f* in the FPGA is applied to each set of data from the five memory banks. After some calculation latency, results from function *f* start filling the result memory bank and the contents of the result memory bank is transferred to allocated CPU memory.

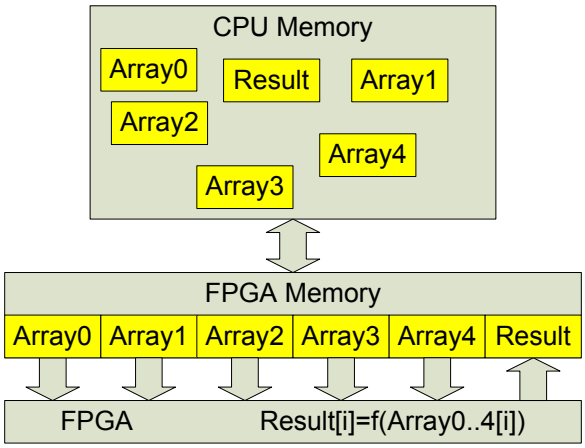

Figure 6 - FPGA and Memory Example

The FPGA and FPGA memory are speed balanced, that is, the FPGA can read any number of attached memory banks simultaneously. In our example in Figure 6, each memory bank is 8 bytes (64 bits) wide and the FPGA in the MAP® Series C operates at 100 MHz (10 nS). This means that with five memory banks used to supply input data to the FPGA, the input data bandwidth to the FPGA is 4 GB/s.

In our ongoing image distance transform example design started in section 1.4, the CPU memory will have several arrays allocated as shown in Figure 7.

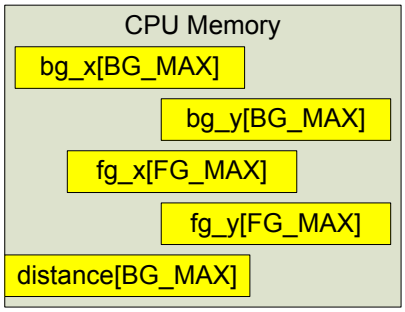

Figure 7 - Distance Transform CPU Memory

Each result produced by the FPGA and stored in the distance array is some function (described in section 1.4) of the two background and two foreground arrays. Since the input arrays are made up of 32 bit sized elements, they may be packed two to a 64 bit container. We will need three of the six FPGA memory banks as shown in Figure 8. All the background pixels' *x* and *y* coordinates are packed into FPGA memory bank A, all the foreground pixels' *x* and *y* coordinates are packed into FPGA memory bank B, and half of FPGA memory bank C is reserved for the shortest distance results.

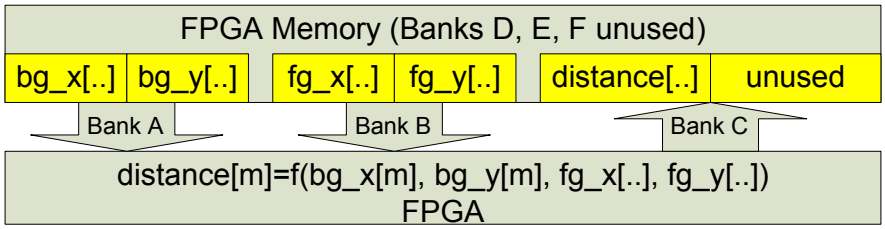

Figure 8 - Distance Transform FPGA Memory

This is why understanding the data movement for an algorithm is important in reconfigurable system design: the programmer is responsible for programming the data movement to and from the FPGA memory. Also, different systems have different bank counts and perhaps even different bit widths for that system's FPGA memory. A programmer has to be keenly aware of the FPGA memory architecture for their target reconfigurable system.

2.3 FPGA and Memory Overhead

In our distance transform example, we've already modified our algorithm because of FPGA memory considerations. The original program description from section 1.4 has become

```
long bg_x[BG_MAX], bg_y[BG_MAX]
long fg_x[FG_MAX], fg_y[FG_MAX]
float distance[BG_MAX]

pack bg_x and bg_y array elements into BG_MAX elements of a 64 bit array
transfer resulting packed array to FPGA memory bank A
pack fg_x and fg_y array elements into FG_MAX elements of a  64 bit array
transfer resulting packed array to FPGA memory bank B
start FPGA processing and wait for completion
```

```
transfer and unpack lower 32 bits from FPGA memory bank C into distance array
```

A reasonable question at this point is why we didn't rework our algorithm into something like

```
int m
long bg_x[BG_MAX], bg_y[BG_MAX]
long fg_x[FG_MAX], fg_y[FG_MAX]
float d, distance[BG_MAX]

pack fg_x and fg_y array elements into FG_MAX elements of a  64 bit array
transfer resulting packed array to FPGA memory bank B
for m = 0 to BG_MAX-1
        transfer bg_x[m] and bg_y[m] to FPGA memory bank A
        start FPGA processing and wait for completion
        transfer and unpack lower 32 bits from FPGA memory bank C into distance[m]
```

The reason that we want to combine the algorithm inner and outer loops into one FPGA function is because of the FPGA and memory overhead (Figure 9) inherent in reconfigurable systems. Every time the FPGA processing is invoked from the program executing on the CPU side, there is a system-dependent time cost that consists of the function call overhead and data movement overhead. In our questionable algorithm where the inner loop "calculate_distance" function is replaced by, the first call to the FPGA is expensive in terms of time: milliseconds from the first function call overhead and the foreground array transfer time. Any calls to the FPGA thereafter cost less time, certainly, but remember that this FPGA function will be called for each background pixel. If the function call overhead is 1 microsecond and there are 1,000,000 background pixels, we've wasted one full second of processing time in unnecessary function call overhead. The goal of the application programmer is to get an application speedup and every possible time-saving trick counts. This is why it is best to port both the algorithm inner and outer loops to FPGA, run the FPGA processing once in order to minimize the time wasted in function call overhead.

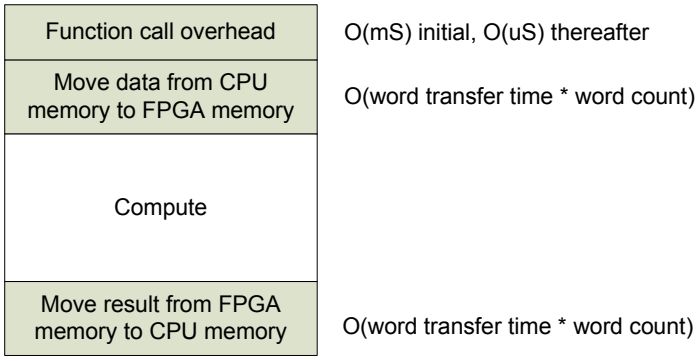

Figure 9 - FPGA and Memory Overhead

Also, in general, it is good practice to move all the data needed by the FPGA computations from CPU memory once and move all of the computational results back to CPU memory once. The more time an algorithm spends computing relative to overhead time, the better the algorithm performance acceleration.

2.4 FPGA Memory Considerations

Our image distance transform example fits well into the SRC MAP® Series C FPGA memory architecture, but what about applications with function operands that do not fit exactly into six 64 bit wide memory banks? Consider Figure 10, where we have an algorithm that produces two 64 bit results from two different functions of seven 64 bit input arrays. All of arrays 0, 1, 2, and 6 are stored before arrays 3, 4, and 5.

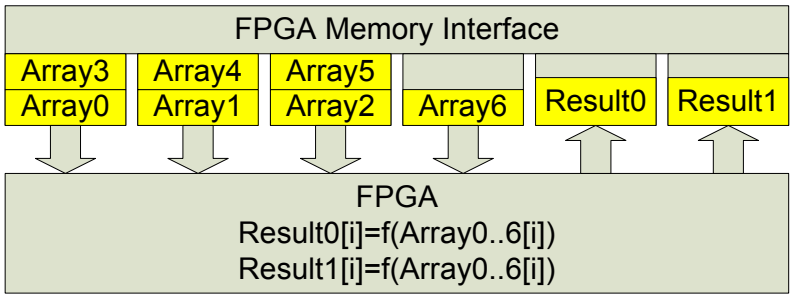

Figure 10 - Difficult FPGA Memory Data Placement

With the CPU data to FPGA memory mapping shown here, all of the required function operands are not simultaneously available to the FPGA. In fact, all data elements of four of the arrays, 0, 1, 2, and 6, must be clocked into and stored in the FPGA before the remaining three arrays, 3, 4, and 5, become available to the FPGA. This memory mapping will likely yield an application slowdown.

Sometimes using the CPU to interleave the operand data array elements in CPU memory first and then transferring the interleaved data to the FPGA memory may yield an overall application performance improvement. The data are interleaved so that the first element of each array is striped across the available FPGA memory banks; the second element of each array is added next, and so on. This results in an FPGA memory layout shown in Figure 11, where all of the data is available for a single result every two clock cycles.

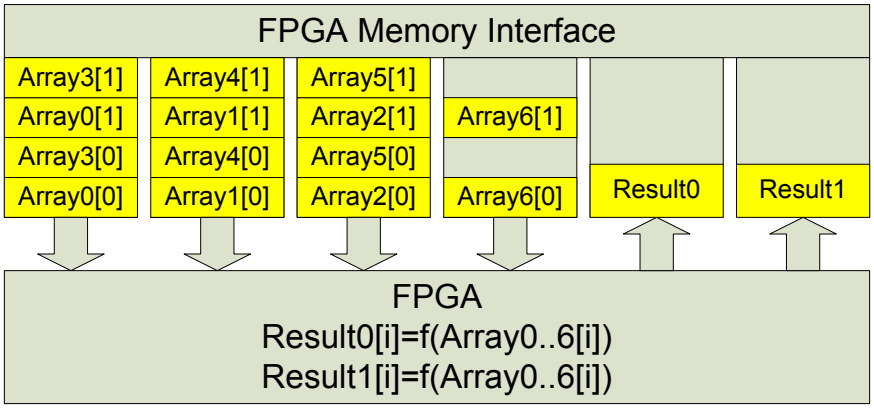

Figure 11 - Interleaved Operand Data Arrays

Another consideration arises if the original CPU application code contains data structures instead of data arrays. These data structures must be "flattened" before transfer to FPGA memory because the FPGA has no method to index into structure elements stored in FPGA memory.

3 Application Metrics

With our work with scientific applications and real world data sets on reconfigurable systems, we observe a set of qualitative metrics that tend to indicate whether or not an application's algorithm is a good candidate for porting to a reconfigurable architecture.

3.1 Operations per data

This metric is an indicator of the potential performance acceleration for an algorithm. In general, the more calculations performed on a given set of data, the better. The central idea here is that the time required to do the work on the data should exceed the time required to move the data to the FPGA memory. Put another way, if a programmer has a design that spends most of its time moving data to and from the FPGA rather than computing, they will not realize a performance gain over CPU-only processing.

For example, consider the algorithm function

$$result[i] = a[i] + b[i] \tag{2}$$

This could be implemented in an FPGA, of course, but consider the amount of data movement relative to the operations. There are two input data arrays, one output data array, and one operation. The bulk of the total FPGA processing time will be spent moving the input arrays to FPGA memory and moving the output array back to CPU memory. There will likely be no performance gain relative to CPU-only execution of this function.

A much better candidate algorithm for FPGA performance acceleration has many operations per unit data. Even better is a set of calculations which has intermediate results that undergo further calculations before the final result. For example, consider

$$\begin{aligned} temp[i] &= \sqrt{(ax[i] - bx[i])^2 + (ay[j] - by[j])^2} \\ result\_a[i] &= temp[i] * 10 \\ result\_b[i] &= \arccos(temp[i]) \end{aligned} \tag{3}$$

This is an excellent candidate for FPGA based performance acceleration. We have four input data arrays, two output data arrays, one intermediate result, and eight operations. The SRC compiler will take care of pipelining the calculations, of course, but this also illustrates the advantage of hardware design: *temp*[*i*], *result_a*[*i*], and *result_b*[*i*] will result in three separate, independent sections of hardware whose execution is fully overlapped. That is, calculations for *result_a*[*i*], *result_b*[*i*], and *temp*[*i+1*] will start as soon as *temp*[*i*] is produced. This sort of parallelism is literally impossible using scalar CPU-only execution: calculations for the *result_a and result_b* arrays may not start until all of the *temp* calculations are performed and the intermediate results pooled.

The high operation per unit data metric for our distance transform algorithm (equation 1) of four input arrays, one output array, and six operations imply that we will achieve some performance improvement with this algorithm using an FPGA.

3.2 Results per data and data reuse

Given that many operations for an amount of data is a good thing for FPGA performance acceleration, multiple results for a set of input data will also likely yield performance improvement. For instance, if result array *X* is a function of the input arrays *A*, *B* and *C*; and result array *Y* is a function of the input arrays *A*, *B*, and *D*, then the performance gains, if any, will result from the simultaneous hardware generation of the two sets of results from the four input arrays.

Another aspect of data reuse is reusing some portion of the input data sets many times for result calculations. Our image distance transform is an example of this aspect of data reuse. Remember that our distance transform algorithm is: for each background pixel, calculate the distance to all foreground pixels and return the least distance. For all result calculations, the background pixel coordinate list is used once and the foreground pixel coordinate list is reused for each background pixel. The data movement from CPU memory to FPGA memory overhead cost is only the time it takes to move the foreground and background coordinates lists once.

3.3 Data per latency

In section 1.2, we introduced the idea of pipelining as a method of exploiting concurrency in an algorithm using reconfigurable computing. It is worth repeating here that the time required to process data in a pipeline is O(*N*+*L*), where *N* is the number of data units to be processed and *L* is the number of stages in the processing pipeline. If *N* is large relative to *L*, then the processing time approaches O(*N*). This means that processing 1,000,000 units of data through a 160 stage pipeline is much better in terms of overall performance than processing 100 units of data through the same pipeline.

Remember that our distance transform algorithm is: for each background pixel, calculate the distance to all foreground pixels and return the least distance. As will be shown in section 5, the hardware pipeline for equation 1 has 90 stages. If we assume a small image size of 512x512 pixels with 131072 foreground pixels and 131072 background pixels, then there will be roughly 1.7×10^{10} pixel coordinate data points used in the calculations. From a qualitative data per latency point of view, this distance transform algorithm is a good candidate for FPGA performance acceleration.

3.4 Will the algorithm fit?

This concept comes from the idea of gaining algorithm performance improvements from both deep and wide parallelism (section 1.2). Implementing deep processing pipelines, or implementing multiple processing pipelines will consume FPGA resources very quickly. The only useful method we've found that addresses this question is to implement a single algorithm processing pipeline for a target FPGA and then estimate how many duplicate pipelines might fit. We encourage programmer experimentation.

A word of caution to FPGA programmers used to the virtually infinite resources available in today's CPU based compute systems: do *not* code up the entire application first and then expect it to fit in a target FPGA. We find that it better to approach FPGA programming in a step-wise fashion, progressing from very basic to more and more complex implementations. Even simple algorithm coding benefits from five or six steps to the final implementation.

Remember that FPGAs in today's reconfigurable systems have improved a great deal over even two and three years ago. Just a few years ago it was difficult to fit a single floating point operation into an FPGA. Today, most or all of a single floating point algorithm from an application will likely fit into an FPGA. Indications from established and startup FPGA vendors confirm that this trend will continue at roughly the same rate or better.

As we develop the implementation of our example distance transform algorithm in section 5, we will discuss the FPGA resource utilization at each development step.

3.5 Table driven calculations

FPGAs are quite efficient at look-up table functions. These tables map well to FPGA chip architectures and produce results within a single clock cycle.

For example, consider the electrostatic interaction calculation between two atoms, one of the five force calculations used in the NAMD molecular dynamics simulation application[1]:

$$U_{Coulomb} = \sum_{i} \sum_{j>i} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \qquad (4)$$

Given a charge q for each atom type in your NAMD simulation, it is easy to build a look up table that yields a $q_i q_j / 4\pi$ value given two atoms i and j. It is easy to see that pre-calculated tables will yield values more quickly than will a calculation engine for this sort of equation. Also, a lookup table-based implementation will likely occupy less space than the actual calculation. Even better from a performance point of view, for a given NAMD simulation run, these look up tables values may be calculated and downloaded to the FPGA before the actual simulation run begins.

3.6 CPU cache breakers

In general, any application with data sets that break CPU cache data availability is a good candidate for reconfigurable computing. This includes data sets that are widely scattered in main memory and data set sizes larger than cache sizes.

An example of this is an output histogram application with a large number of result bins. The output bin values are updated in an unpredictable order. As the histogram bin count grows, it becomes more and more likely that any given bin value will not be available in the cache when it is needed, which results in a large number of high latency main memory accesses. On the other

[1] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale, and K. Schulten, *Scalable Molecular Dynamics with NAMD*, Wiley Interscience (www.interscience.wiley.com), 26 May 2005.

hand, the nature of FPGA programming dictates that all data is local to the FPGA before processing, and so, available when it is needed.

4 Example Application Target System: SRC MAPstation

In Section 5, we will work through several development iterations of our image distance transforms implementation on an FPGA. But before we get to the good stuff, since different reconfigurable systems have different architectures that must be understood by the programmer, we will first discuss the target machine for this example: the SRC MAPstation with a MAP® Series C FPGA module. We also introduce the SRC Carte™ programming environment.

4.1 System Architecture

We will consider the overall system architecture of the SRC MAPstation, then the detailed architecture of the MAP® Series C module that contains the FPGAs and FPGA memory.

4.1.1 SRC MAPstation

The SRC MAPstation (Figure 12) used for our example contains a commodity dual CPU board, a MAP® Series C processor, and an 8 Gbyte common memory module, all interconnected with a 1.4 Gbyte/s low latency switch.

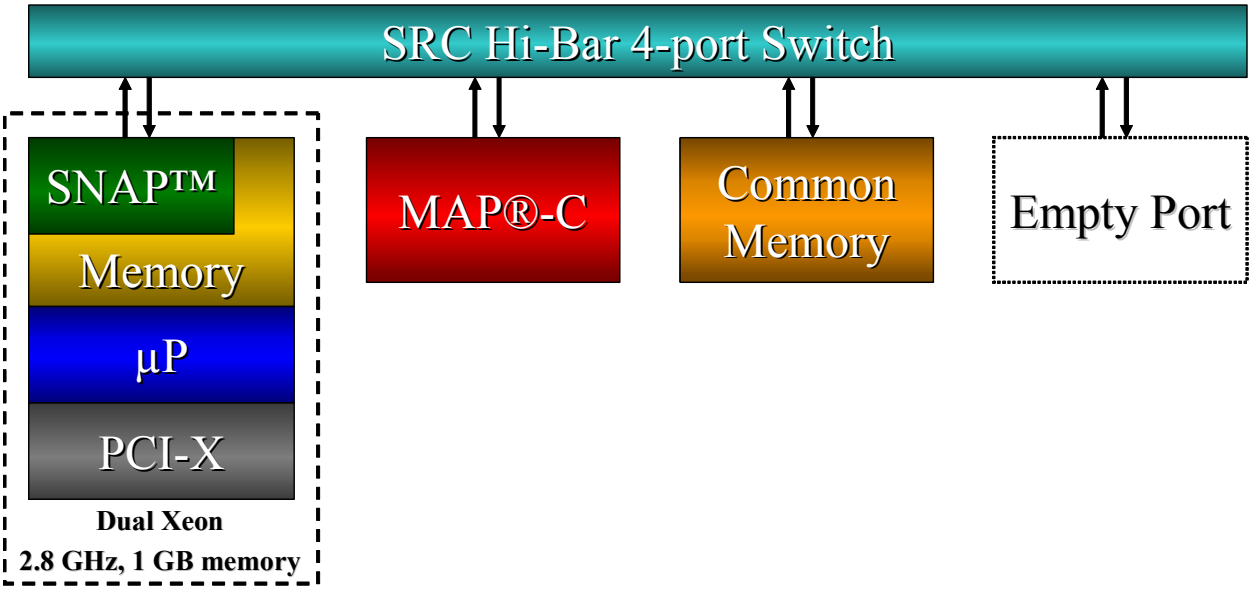

Figure 12 - SRC MAPstation Architecture

The SNAP™ Series B interface board is used to connect the commodity dual CPU board to the Hi-Bar switch. The SNAP™ plugs directly into a CPU board's DIMM memory slot so that it can sustain a much higher data bandwidth than can the CPU's I/O subsystem – roughly four times higher sustained bandwidth than that available from 133 MHz PCI-X.

While our system has the common memory module installed, it is not used in our example. Applications on the SRC MAPstation might typically use this common memory as large intermediate data storage between the CPU and FPGA memories.

4.1.2 MAP® Series C Processor Module

The MAP® Series C processor module (Figure 13) contains the two user FPGAs and the FPGA memory among other things that are not important for the purpose of this discussion.

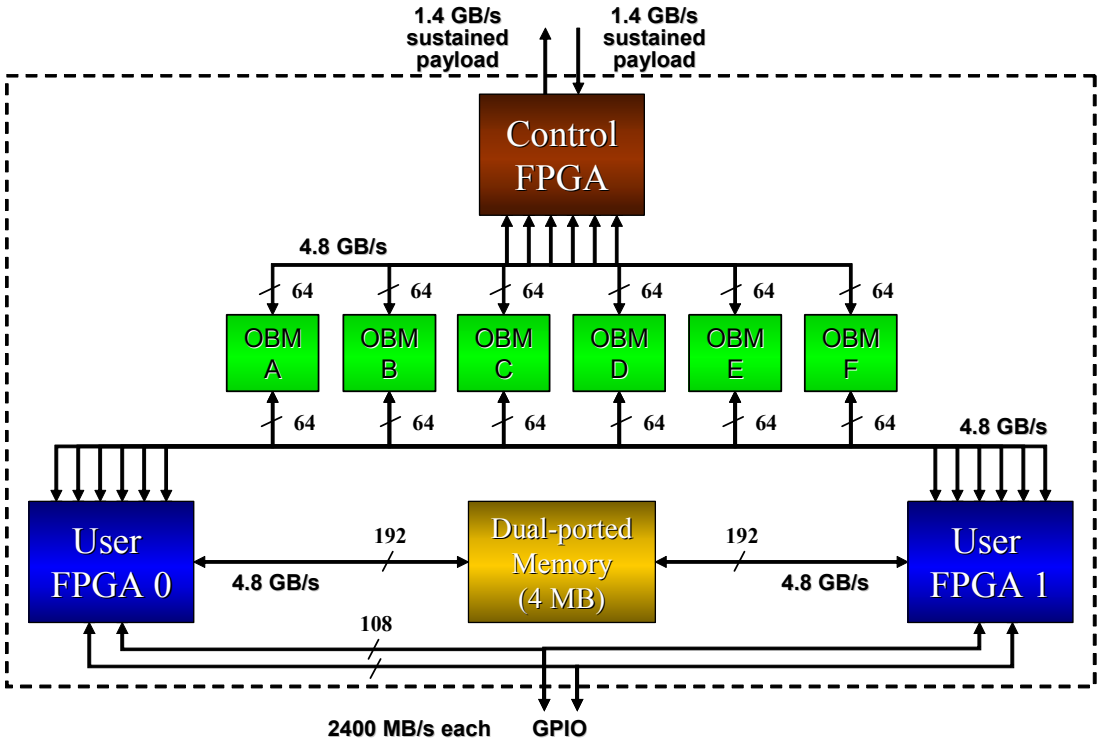

Figure 13 - SRC MAP® Series C Architecture

There are six banks of MAP® Series C on-board memory (OBM A-F). Each bank is 64 bits wide and 4 Mbytes deep for a total of 24 Mbytes. The programmer is responsible for application data transfer to and from these memories via the use of SRC programming macros invoked from the FPGA application. All six of these on board FPGA memories are available to both of the user FPGAs, although not at the same time. The OBM to user FPGA mapping and transfer of control is also the responsibility of the programmer, through the use of SRC programming macros.

There is an additional 4 Mbyte of dual-ported memory dedicated solely to data transfer between the two FPGAs. This is not directly visible to the programmer, but utilized through the use of SRC programming macros.

The control FPGA is completely invisible to the programmer. It provides the interface logic between the Hi-Bar switch and the MAP® Series C module.

The two user FPGAs in the MAP® Series C are Xilinx Virtex-2 V6000 FPGAs. They each contain 6 million equivalent logic gates, 144 dedicated 18x18 integer multipliers, and 324 Kbytes of internal dual-ported block RAM (BRAM).[1] These FPGA elements are not directly visible to the programmer, but are interconnected appropriately as determined by the programmer's 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.

Our distance transform algorithm example in section 5 starts off with a CPU-only design, moves on to a single FPGA design, then finally describes an implementation that utilizes both FPGAs in the MAP® Series C.

4.2 SRC Carte™ Programming Environment

The programming environment (Figure 14) 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 CPU code and FPGA hardware programming files. The debug version is useful for code testing before the final time intensive hardware place and route step. Intel icc compiler is used to generate both the CPU-only debug executable and the CPU-side of the combined CPU/FPGA executable.

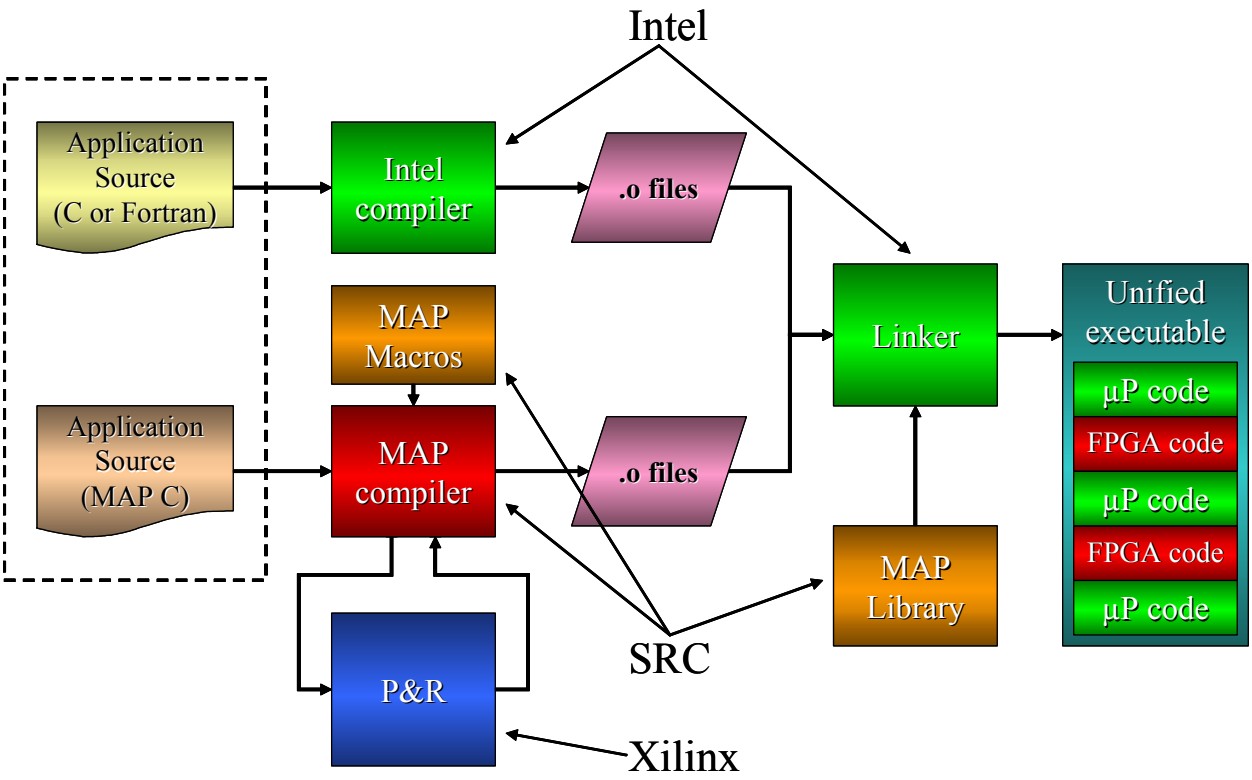

Figure 14 - SRC Carte™ Programming Environment

[1] Xilinx, Inc., Virtex II Platform FPGAs: Complete Data Sheet, DS031 (v3.4), San Jose, CA, March 1, 2005, page 2.

The SRC MAP® compiler is invoked by the makefile to produce the hardware description of the FPGA design for final combined CPU/FPGA 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.

5 Image Distance Transform Implementations

Image distance transform is an operation that is typically performed on binary images. The result of the transform is a gray scale image in which each background pixel is replaced with the shortest distance to the foreground pixels. Figure 15 shows an example of a binary image and its distance transform. It is often used in image processing applications to isolate a portion of an image for further processing.

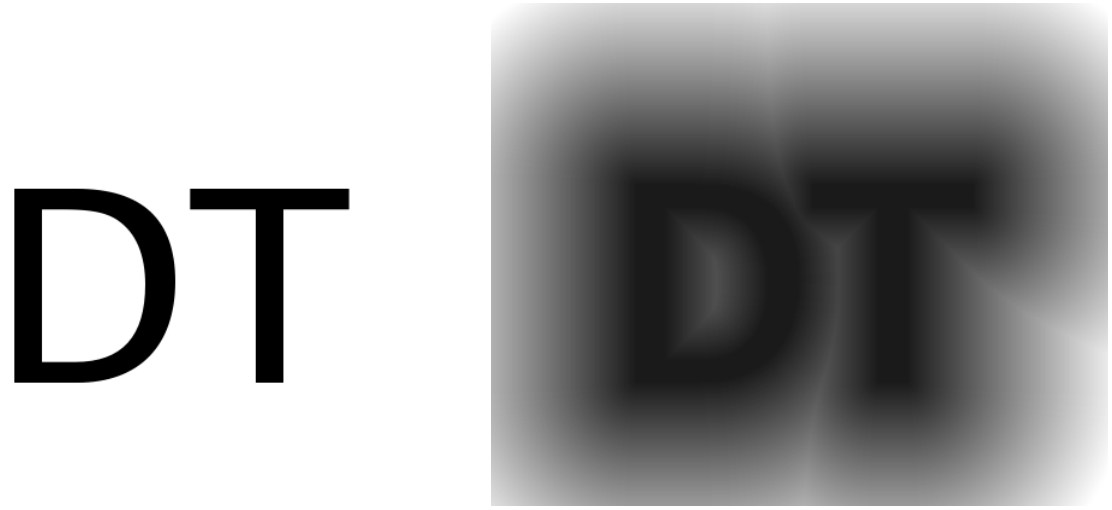

Figure 15 – Initial (left) and Distance Transformed (right) Images

The simplest (and perhaps most inefficient) algorithm to compute the distance transform is as follows: for each background pixel compute distances to every foreground pixel and use the shortest found distance as the gray scale value that replaces that background pixel. Assuming that half of the pixels belong to the foreground and half of the pixels belong to the background, the overall compute time is $O(N^2)$, which is the worst-case scenario as far as the ratio of the foreground and background pixels is concerned.

5.1 Algorithm Performance Considerations

Before moving on to the example algorithm development, we need to define and discuss the performance measurement details. Figure 16 shows the three time components of the base line algorithm execution on a CPU: getting the image ready for the distance transform, actually computing the distance transform for each background pixel against all foreground pixels, and then the transformed image assembly.

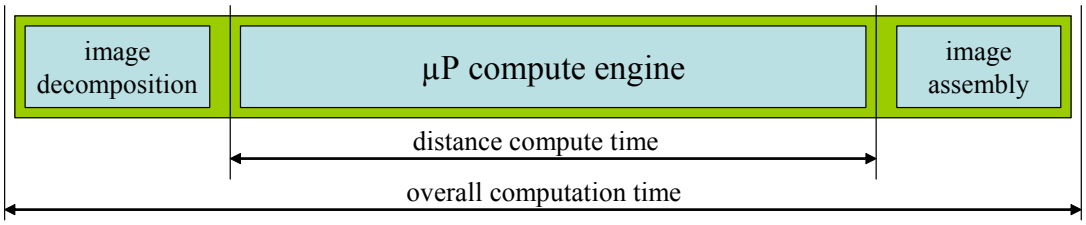

Figure 16 - CPU-only Algorithm Performance Measurement

For the combined CPU and FPGA algorithm performance measurement (Figure 17), the image decomposition and image assembly times are identical to the CPU-only measurements, but the distance transform compute time on the FPGA has four components: the transform compute time plus the FPGA overhead discussed in section 2.3. In general, CPU vs. CPU/FPGA performance numbers needs to be examined with some care since some published algorithm performance numbers only compare the distance compute time of CPU vs. FPGA and leave out the FPGA usage overhead. We are concerned here with actual system performance comparisons, as these complete performance numbers indicate the real benefit that a programmer may realize with a reconfigurable system.

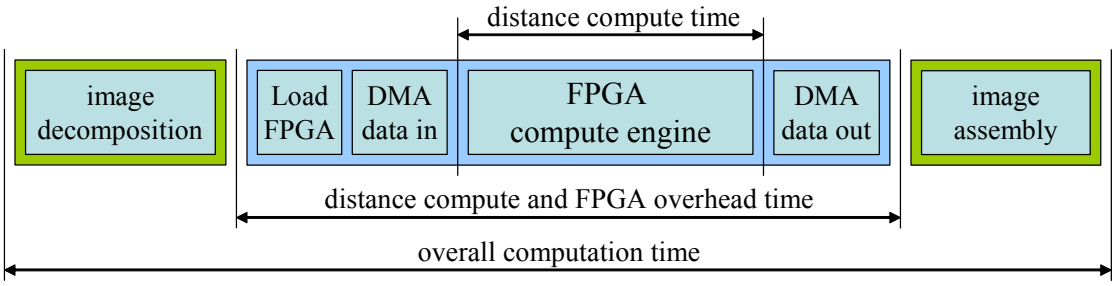

Figure 17 - Combined CPU/FPGA Algorithm Performance Measurement

5.2 CPU implementation

Let us first consider a CPU implementation of the brute-force algorithm described in the previous section. Figure 18 shows our image pixel row and column coordinate system for this implementation. There are up to *m* rows and *n* columns of pixels. For convenience, we will map the two dimensional image array of pixels into a one dimensional array $img[]$ such that a single pixel $img[i, j]$ may be indexed as $i * m + j$.

Distance calculations require pixel coordinates; therefore, we need to assemble separate lists of pixel coordinates i, j belonging to the foreground and background pixels. There can be no more than $MAXP = m * n$ foreground or background pixels. Therefore, we can allocate two arrays containing $2 * MAXP$ pixel coordinates each, which will be large enough to hold any possible number of foreground and background pixel coordinates for the image of size *MAXP* pixels. How much memory should we allocate for each pixel? That depends on the size of the image: if the image size is less than 65536 pixels in both the *i* row and *j* column dimensions (and

65536x65536 pixels is a HUGE image!), then 'short int' is sufficient. The background or foreground pixel i, j coordinates are stored in the appropriate list i first, then j.

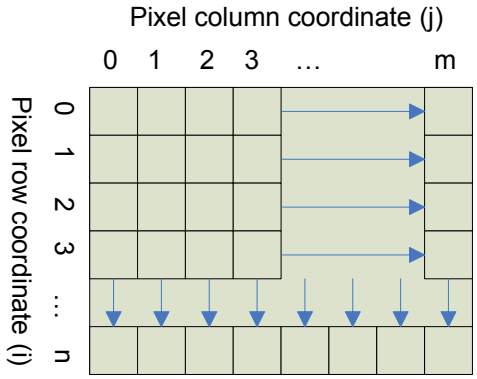

Figure 18 - Image Pixel Coordinates (img[i*m+j])

Now we know all we need to allocate the memory for the lists of the pixel coordinates and to assemble the lists:

```
// allocate memory for the foreground/background pixel coordinate lists
short *fg_pixel = (short*)malloc(2*MAXP*sizeof(short));
short *bg_pixel = (short*)malloc(2*MAXP*sizeof(short));

// extract background and foreground pixels
int fgc = 0;        // foreground pixels coordinate count
int bgc = 0;        // background pixels coordinate count

for (i = 0; i < n; i++)                    // for each pixel row i in image
{
        for (j = 0; j < m; j++)            // for all pixel columns in row i in image
        {
                if (img[i*m+j] == FOREGROUND)    // if this is a foreground pixel
                {
                        fg_pixel[fgc] = i;              // store [i,j] pixel location in fg_pixel list
                        fg_pixel[fgc+1] = j;
                        fgc += 2;
                }
                else                                    // else is a background pixel
                {
                        bg_pixel[bgc] = i;              // store [i,j] pixel location in bg_pixel list
                        bg_pixel[bgc+1] = j;
                        bgc += 2;
                }
        }
}
```

Note that at the end of this procedure, we know how many foreground and background pixels are stored: $fgc/2$ in the foreground list and $bgc/2$ in the background list.

Now we can allocate memory for the computed distance results. We have $fgc/2$ foreground pixels in total and we should use 'float' as the result data type so that we do not compromise accuracy in case if any follow-up calculations are to be performed:

```
float *bg_distance = (float*)malloc((fgc/2)*sizeof(float));
```

At this point we are ready to perform the required distance calculations. Also, since this is the computational core of our algorithm, we will measure how much time it takes to perform these calculations:

```
gettimeofday(&t0, NULL);

for (i = 0; i < bgc; i += 2)                    // for each image background pixel
{
        short x = bg_pixel[i];
        short y = bg_pixel[i+1];
        long d_min = 1000000;

        for (j = 0; j < fgc; j += 2)            // store the distance to the nearest foreground pixel
        {
                short dx = x - fg_pixel[j];
                short dy = y - fg_pixel[j+1];
                long d = dx * dx + dy * dy;
                if (d < d_min)
                        d_min = d;
        }
        bg_distance[i/2] = sqrt(d_min);         // new gray scale value for this background pixel
}

gettimeofday(&t1, NULL);
```

The CPU distance calculation compute time is now simply:

```
t1.tv_sec - t0.tv_sec + (t1.tv_usec - t0.tv_usec) * 1e-6
```

We still have to assemble the resulting image and free memory used in the process and free unneeded memory:

```
// add foreground pixels to the resulting image unchanged
for (k = 0; k < fgc; k += 2)
        result_img[fg_pixel[k]* m+fg_pixel[k+1]] = FOREGROUND;

// add background pixels as the distance to the nearest foreground pixel
for (k = 0; k < bgc; k += 2)
        result_img[bg_pixel[k]* m+bg_pixel[k+1]] = (short) bg_distance[k/2];

free(fg_pixel);
free(bg_pixel);
free(bg_distance);
```

As we look at the code structure, it consists of image decomposition into the foreground and background pixel lists, distance transform, and image assembly as shown in Figure 16 ("overall computation time"). How quickly does the "distance compute time" of this CPU-only code run? Figure 19 shows the compute time as a function of the number of foreground pixels for an image of 512x512 pixels. As expected, time to compute will be the longest when one half of the pixels belong to the background and another half belongs to the foreground. Perhaps a more desirable plot would be to show time to compute as a function of total number of distance calculations, which will effectively eliminate half of the plot shown in Figure 19. We will return to this idea later as we port the code to FPGA and measure its performance.

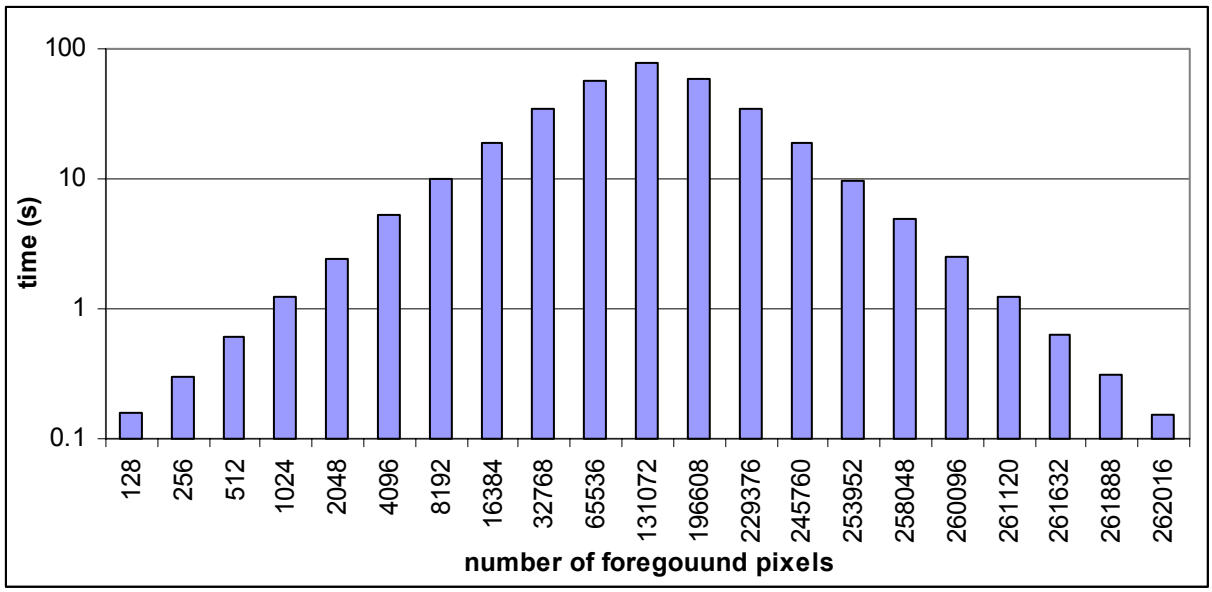

Figure 19 – CPU Distance Compute Time

What about time spent to decompose the image into background/foreground pixel and to lists and assemble the resulting image (the "overall computation time" minus the "distance compute time" of Figure 16)? It turns out that these two procedures are very inexpensive since the number of operations needed to perform these procedures is lineal in the image size space and it does not require any expensive calculations – it is basically moving data in the CPU memory. For example, for the 131072 foreground pixels case the execution time (wall clock time) of the entire code is 78.367 seconds whereas the execution time of the nested loop part of the code is 78.352 seconds. Therefore, in this particular application we can safely ignore the CPU memory data movement overhead due to these operations. We encourage the programmer to keep in mind that this may not be the case in other applications.

5.3 Easy FPGA implementation

This is not meant to be a tutorial of how to use the SRC development tools and libraries; rather, it is an illustration of application programming. We assume that the programmer has some familiarity with the SRC MAPstation and Carte™ Programming Environment documentation.

The first question that we need to answer is how to partition the application code between CPU and FPGA. Since we already have done timing analysis of the CPU-only code and know that the image decomposition and image assembly operations are fast and the distance calculations nested loop is slow, then perhaps we should port the nested loop code segment to the FPGA and keep everything else on the CPU.

The second question that we need to answer is how to deal with data movement between the FPGA memory and the CPU. We need to supply FPGA with the coordinates of the foreground and background pixels and we expect the MAP® Series C processor to return a list of distances (Figure 20) which we can then re-map into a result image. Note that the distance list should be in the same order as the list of the background pixels, otherwise we will not be able to properly reconstruct the output image.

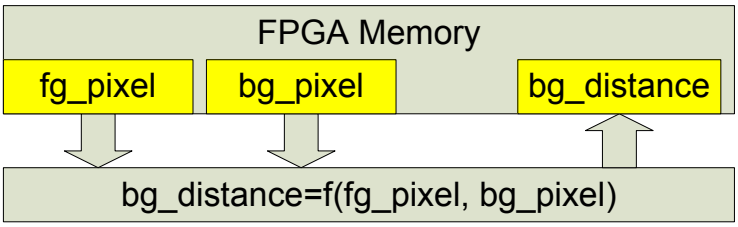

Figure 20 - Easy FPGA Implementation Memory Layout

Fortunately for us (or was that by design?), foreground and background pixel coordinates are already placed into 2 one-dimensional arrays and each pixel coordinate value occupies exactly 4 bytes (short int). This yields 2 sets of pixel coordinates per 64 bit word, which is the base data type in the MAP® Series C processor. Therefore, on the CPU side of the code we can simply cast the fg_pixel and bg_pixel pointers to int64_t*. Also, for performance reasons, we need to use different memory allocation functions:

```
short* fg_pixel = (short*)Cache_Aligned_Allocate(MAXP*sizeof(short));
short* bg_pixel = (short*)Cache_Aligned_Allocate(MAXP*sizeof(short));
```

The same is true for the resulting distances: they are stored as a one-dimensional array of single precision floating point numbers, 2 pixel coordinate values per 64 bit word:

```
float* bg_distance = (float*)Cache_Aligned_Allocate((bgc/2)*sizeof(float));
```

Now we are ready to replace the nested loop code in the CPU code with a MAP® function call:

```
int64_t tm1, tm2, tm3;    // will be used to store timing measurements on the FPGA
int mapnum = 0;

gettimeofday(&t0, NULL);

dtransform_hw((int64_t *)fg_pixel, (int64_t *)bg_pixel, (int64_t *)bg_distance,
                    fgc/2, bgc/2, &tm1, &tm2, &tm3, mapnum);

gettimeofday(&t1, NULL);
```

Our job on the CPU side is done. (Note that map_allocate() needs to be called before dtransform_hw() and map_free() should be called after, but you already knew this from reading the SRC documentation, right?)

At this point we can get some estimates about the FPGA code performance, even without implementing the code! We know that we have $(fgc + bgc)/2$ pixel coordinate pairs to transfer into MAP® Series C memory and $bgc/2$ distances to transfer back to CPU memory, that's $S = ((((fgc + bgc)/2) \cdot 2 \cdot szi) + (bgc/2 \cdot szf))$ bytes in total, where *szi* is sizeof(short int) and *szf* is sizeof(float). If we choose to use the striped DMA supported by the SNAP™ interface (section 4.1.1), in one clock cycle we can transfer two 64 bit words (16 bytes) between CPU memory and MAP memory. Therefore, the DMA data transfer will require $S/16$ clock cycles, or $S/16 \cdot 10^{-9}$ seconds with a 100 MHz FPGA clock. This is the total "DMA Data In" plus "DMA Data Out" time in Figure 17.

Where do we put all the pixel coordinates in MAP memory? This depends on the processing algorithm to be implemented on the MAP and on the overall image size. For simplicity, let us first directly port the code as is, that is, implement just one calculation per loop. In this case, we can put the foreground pixels to OBM bank A, background pixels to OBM bank B, and resulting distances to OBM band C (Figure 20). This way the FPGA will be able to access both the source and destination FPGA memory in one clock cycle. There is enough FPGA memory to store input coordinates of 1048576 pixels in each bank and store 1048576 computed distances in bank C. This translates into an image size of, say, 1024x1024 pixels that we can process by just using 3 OBM memory banks. If a larger image needs to be processed, the remaining OBM banks can be used.

Here is our MAP® function call that implements our nested loop code segment derived from the CPU-only code:

```
void dtransform_hw(int64_t fg_pixel[], int64_t bg_pixel[], int64_t bg_distance[],
                   int32_t fg_count, int32_t bg_count,
                   int64_t *tm1, int64_t *tm2, int64_t *tm3, int mapnum)
{
        // declare FPGA memory
        OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)    // foreground pixel coordinate list
        OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)    // background pixel coordinate list
        OBM_BANK_C (CL, int64_t, MAX_OBM_SIZE)    // distance result list

        int i;                          // background pixel list iterator
        int j;                          // foreground pixel list iterator
        int32_t mindA, mindB;       //  minimum estimated distance for this j iteration
        int32_t min_da, min_db;     //  running minimum distance for this i iteration
        int64_t t0, t1, t2, t3;     // timers for "distance compute and FPGA overhead" in Figure 17

        read_timer(&t0);    // "DMA data in" (Figure 17) section starts here

        // transfer foreground pixels from CPU memory to FPGA memory
        DMA_CPU(CM2OBM, AL, MAP_OBM_stripe(1,"A"), fg_pixel, 1, fg_count*4, 0);
```

```
        wait_DMA(0);

        // transfer background pixels from CPU memory to FPGA memory
        DMA_CPU(CM2OBM, BL, MAP_OBM_stripe(1,"B"), bg_pixel, 1, bg_count*4, 0);
        wait_DMA(0);

        read_timer(&t1);    // "distance compute time" (Figure 17) section starts here

        for (i = 0; i < bg_count/2; i++)            // for each pixel in the background list
        {
                for (j = 0; j < fg_count/2; j++)    // find the nearest foreground list pixel
                {
                        // calculate the distance estimate for all forground pixels
                        // against the two j-th background pixels and keep a
                        // running minimum distance for both background pixels
                        distance(BL[i], AL[j], &mindA, &mindB);
                        cg_accum_imin_32(mindA, 1, 100000, j == 0, &min_da);
                        cg_accum_imin_32(mindB, 1, 100000, j == 0, &min_db);
                }

                // we have the minimum estimated distance for two background
                // pixels, take the final square root of these and store to OBM C
                comb_32to64_flt_flt(sqrt(min_da), sqrt(min_db), &CL[i]);
        }

        read_timer(&t2);   // "DMA data out" (Figure 17) section starts here

        // transfer out result array
        DMA_CPU(OBM2CM, CL, MAP_OBM_stripe(1,"C"), bg_distance, 1, bg_count*4, 0);
        wait_DMA(0);

        read_timer(&t3);   // end timer for "DMA data out"

        *tm1 = t1 - t0;    // store "DMA data in" time
        *tm2 = t2 - t1;    // store "distance compute time"
        *tm3 = t3 - t2;    // store "DMA data out" time
        }
```

```
    void distance(int64_t pBG, int64_t pFG, int32_t *min_d1, int32_t *min_d2)
    {
            int16_t pbg1y, pbg1x, pbg2y, pbg2x;  // two bg pixels (four coordinates total) from OBM B
            int16_t pfg1y, pfg1x, pfg2y, pfg2x;     // two fg pixels (four coordinates total) from OBM A
            int16_t p1xx, p1yy, p2xx, p2yy;         // estimated distance calculation results
            int32_t d1, d2;                         // minimum estimated distance calculation results

            // split the input data into two bg and two fg pixel coordinate sets
            split_64to16(pBG, &pbg1y, &pbg1x, &pbg2y, &pbg2x);
            split_64to16(pFG, &pfg1y, &pfg1x, &pfg2y, &pfg2x);

            // Calculate the distance estimate (equation 1 without the square root)
            // for the first background pixel and the two foreground pixels, then
            // store the lesser result in min_d1.

            p1xx = pbg1x - pfg1x;
            p1yy = pbg1y - pfg1y;
```

```
            d1 = p1xx * p1xx + p1yy * p1yy;

            p2xx = pbg1x - pfg2x;
            p2yy = pbg1y - pfg2y;
            d2 = p2xx * p2xx + p2yy * p2yy;

            *min_d1 = (d1 < d2) ? d1 : d2;

            // Calculate the distance estimate (equation 1 without the square root)
            // for the second background pixel and the two foreground pixels, then
            // store the lesser result in min_d2.

            p1xx = pbg2x - pfg1x;
            p1yy = pbg2y - pfg1y;
            d1 = p1xx * p1xx + p1yy * p1yy;

            p2xx = pbg2x - pfg2x;
            p2yy = pbg2y - pfg2y;
            d2 = p2xx * p2xx + p2yy * p2yy;

            *min_d2 = (d1 < d2) ? d1 : d2;
    }
```

Since the input pixel coordinates are packed into 64 bit words, each time the FPGA reads one data word from a single OBM bank, it reads coordinates for two consecutive pixels. Therefore, in one clock cycle, the FPGA accesses two OBM banks: two pixels from the foreground list and two pixels from the background list, which allows it to compute 4 distances in parallel.

Square root calculations are pipelined by the SRC compiler libraries for best performance, but still, square root calculation engines are expensive in terms of FPGA resources. Note that the in-lined distance() function calculates four distance estimates (the distance calculation without the square root) and returns only the two minimum distance estimates for that pass. The cg_accum_imin_32() functions keep a running minimum distance estimate for the two background pixels in the *i*-th loop and the final square root calculation is performed after the distance to the nearest foreground pixels has been found for these two background pixels. This allows for the instantiation of only two hardware square root engines in the FPGA rather than four, a substantial savings in FPGA resources.

We take timestamps between various code segments so that we can get precise measurements of the time spent executing one or another section of the FPGA code. Also note that this code will produce correct results only when the foreground pixels list has an even number of pixels in it since we do not perform any boundary checks. This limitation can be fixed either on the CPU or FPGA side by either adding extra check, or by setting and extra pixel by duplicating one of the existing foreground pixels in case of odd foreground pixels list size.

The SRC MAP® compiler shows that the inner loop was successfully pipelined and its pipeline depth is 16:

```
##################        INNER LOOP SUMMARY      ####################
loop on line 33:
```

```
    clocks per iteration:    1
    pipeline depth:         16
########################################################################
```

The Xilinx place & route tools show that the design fit into the MAP® Series C FPGA just fine and meets timing specifications:

```
###############        PLACE AND ROUTE SUMMARY    ####################
  Number of Slice Flip Flops:      14,367 out of  67,584   21%
  Number of 4 input LUTs:          10,070 out of  67,584   14%
  Number of occupied Slices:       10,055 out of  33,792   29%
  Number of MULT18X18s:                24 out of     144   16%
  freq = 100.4 MHz
########################################################################
```

Let's run it and compare its performance to the CPU code performance. Figure 21 shows both the time spent by the CPU and time spent on FPGA for the equal number of computed distances. The best overall speedup of ~1.8 times is achieved when there is at least 1056964608 distances are to be computed (which corresponds to having 4096 foreground pixels).

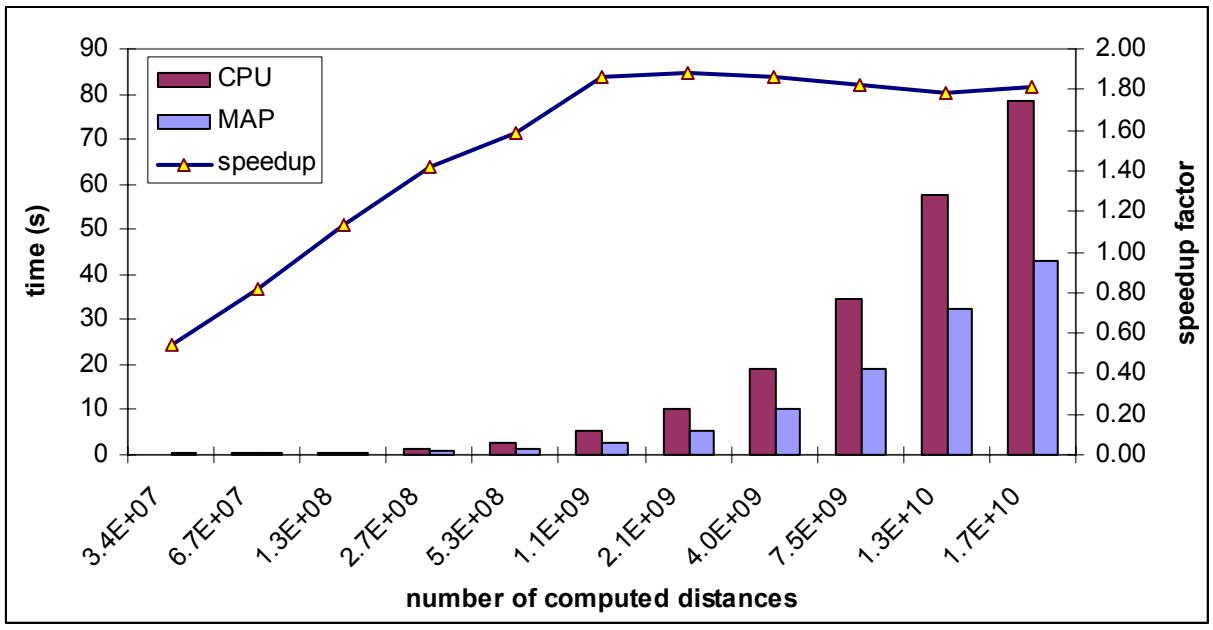

Figure 21 - CPU vs. FPGA Performance (1)

5.4 Improved FPGA Implementation

These results are not particularly exciting: 1.8x performance improvement for all of that work? What went wrong? Nothing, really. This illustrates our point on iterative development we first mentioned in section 3.4. We suggest that a programmer needs to start out with a simple reconfigurable system implementation, examine the results, and work in a step-wise manner towards an acceptable implementation. This is our first simple implementation of the image distance transform algorithm.

We encourage programmers to not be faint of heart: our first successful NAMD implementation on a reconfigurable system had a 200x *slowdown*. This first implementation produced correct numerical results, but the performance results were not hopeful. It took us ten iterations to get to a 3x speedup, and we are still working on this application. The point is that it is important to get that first implementation running correctly at *any* performance, so that step-wise experimentation can guide the programmer as they carefully fit their application onto reconfigurable system architecture.

So let's take our own advice and look at the results of our first implementation of the image distance transform. Let's examine the timing details for the code executed on MAP® Series C with 32768 foreground pixels:

```
------------------------------------------------------------
CPU time (overall)      : 18.986597 seconds
CPU time (compute only) : 18.975332 seconds
------------------------------------------------------------
FPGA time         : 18.909964 seconds (1890996366 clock cycles)
  - data transfer : 0.002500 seconds (249981 clock cycles)
    - in          : 0.001331 seconds (133060 clock cycles)
    - out         : 0.001169 seconds (116921 clock cycles)
  - COMPUTE       : 18.907464 seconds (1890746385 clock cycles)
------------------------------------------------------------
```

A negligible amount of time was spent transferring data in and out, as expected, but quite a bit of time was spent doing actual calculations, 18.91 seconds in total. Why is that? Consider the following: for an image of size 512x512 pixels containing 32768 foreground pixels, there are (512*512-32768)* 32768=7516192768 distances to compute. We perform 4 distance calculations per clock cycle, that's 7516192768/4=1879048192 clock cycles, or 18.79 seconds. The remaining 1890996366-1890746385=249981 clock cycles were spent in outer loop execution, which was not pipelined. Therefore, if we want to achieve a better overall code performance, we need to shrink the FPGA compute time. Since FPGA frequency is fixed, this can only be achieved by performing more calculations at once. But in order to perform more calculations we need to have simultaneous access to more data. What are our options? What do we need to consider?

Unused OBM Memory. There are three OBM memory banks on the MAP® Series C that we have not used. If we put the background pixels in OBM A, use OBM B, C, D, and E for foreground pixels, and OBM F for results, then the FPGA has access to 8 foreground pixels in one clock cycle - we could speedup the calculations by a factor of 4.

Loop Unrolling. We are wasting 11948174 clock cycles for the *i* outer loop iterations. We need to combine the *i* and *j* loop iterations.

Internal FPGA Memory (BRAM). We could utilize the internal FPGA memory, although its small size will limit the maximum size of images that we could process.

FPGA Multipliers. The current design utilizes 16% of the available dedicated hardware multipliers on the FPGA, roughly 4% per calculation. This indicates that we may be able to implement 20 or so distance calculations on a single FPGA in the MAP® Series C.

Second MAP® Series C FPGA. We will get more multipliers by splitting our design across the two user FPGAs in the MAP® Series C. This, however, will prohibit us from using 4 OBM banks for storing the foreground pixels for the first FPGA – the second FPGA will require exclusive use of two OBM banks.

What is the right answer? As always, the answer is "it depends". We encourage experimentation on the part of the programmer. When we get to this point, we depend heavily on trying several ideas and examining the performance results to guide us to a good solution.

After a bit of experimentation, we came up with the following dual FPGA implementation. Half of the background pixels are stored in OBM A, the other half are stored in OBM B. Resulting computed distances are stored in OBM C and D. The primary FPGA uses banks A and C, the second FPGA uses banks B and D, and each chip is responsible for computing half of the results. Foreground pixels are divided into 5 equal groups. Four of these groups are stored in BRAM memory and the fifth in OBM. This fifth group of foreground pixels is duplicated in OBM E and F – OBM E for the primary FPGA and OBM F for the secondary.

Remember that each 64 bit input word contains a pair of pixels (4 coordinates) of data. Thus, for each pair of the background pixels we have simultaneous access to 10 pairs of the foreground pixels on each FPGA chip (4 in BRAM, 1 in OBM times 2 pixels per word = 10 pairs of foreground pixels.) This allows us to perform 20 simultaneous distance calculations on a single FPGA (5 parallel distance calculation engines, 4 parallel calculations per engine), or 40 simultaneous distance calculations on the two FPGAs in the MAP® Series C module.

That's the good news. The bad news is that we have a limit on the maximum number of foreground pixels in an image we intend to process. Our implementation uses 4 banks of internal FPGA (BRAM) memory and 1 OBM bank to hold all of the foreground pixels. We can't split the foreground pixels between the FPGAs, either, remember that all foreground pixels have to be run against any one background pixel. Each of the four BRAM banks may be 64 bits wide, which is fine, but the BRAM array index maximum value must be a multiple of 2 and not exceed the total available BRAM on the FPGA. This allows us to declare four 64 bit arrays of 8192 elements each. This yields 8192 elements per array multiplied by 8 bytes per element multiplied by 4 arrays = 262144 bytes, which does not exceed the maximum 324000 bytes of BRAM available on the Virtex 2v6000 FPGA.

We can store 2 pixels in each 64 bit word, so we can store 8192 elements per BRAM array multiplied by 2 pixels per element = 16384 pixels per array. We have 4 BRAM arrays and 1 OBM array for foreground pixels, so we are limited to images with 81920 foreground pixels or less.

The source code for the primary chip follows; the computational section of the code for the secondary chip is similar. Note that we use the same inlined distance() function introduced in section 5.3 and that the CPU side code for this implementation does not change.

```
void dtransform_hw(int64_t fg_pixel[], int64_t bg_pixel[], int64_t bg_distance[],
                   int32_t fg_count, int32_t bg_count,
                   int64_t *tm1, int64_t *tm2, int64_t *tm3, int mapnum)
{
    // declare FPGA on-board memory
    OBM_BANK_A (AL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_B (BL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_C (CL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_D (DL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_E (EL, int64_t, MAX_OBM_SIZE)
    OBM_BANK_F (FL, int64_t, MAX_OBM_SIZE)

    // declare internal FPGA memory (BRAMs)
    int64_t fg_pix01[8192];
    int64_t fg_pix02[8192];
    int64_t fg_pix03[8192];
    int64_t fg_pix04[8192];

    int64_t v0;                 // primary and secondary FPGA common parameters
    int 64_t count;             // iterator used for splitting up the foreground pixels
    int 64_t indx;              // index used to determine split of foreground pixels
    int 64_t k;                 // main computational loop counter
    int 64_t nofiterations;     // main computational loop maximum count
    int64_t t0, t1, t2, t3;     // sectional performance timers
    int32_t i, j;               // image row and column indices

    // pairs (a,b) of minimum distances calculated by distance() function number 1-5
    int32_t mind01a, mind02a, mind03a, mind04a, mind05a;
    int32_t mind01b, mind02b, mind03b, mind04b, mind05b;

    // pair (a,b) of running minimum estimated distances
    int32_t mindAa, mindAb;

    // pair (a,b) of minimum estimated distances
    int32_t min_da, min_db;

    read_timer(&t0);    // “DMA data in” (Figure 17) section starts here

    // sync primary and secondary FPGA, no OBM permissions granted at this time
    send_perms(0);

    // send common parameters to secondary FPGA
    comb_32to64(fg_count, bg_count, &v0);
    send_to_bridge(v0);

    // copy the foreground pixel data from CPU memory to FPGA on board memory
    DMA_CPU(CM2OBM, AL, MAP_OBM_stripe(1,"A,B,C,D,E"), fg_pixel, 1, fg_count*4, 0);
    wait_DMA(0);

    // split the foreground pixels into five pieces: four to BRAM
    // and one to OBM E.
```

```
count = ((fg_count % 10) ? fg_count+10 : fg_count)/10;
for (k = 0; k < count; k++)
{
        indx = k * 10;
        fg_pix01[k] = AL[k];
        fg_pix02[k] = (indx+2 >= fg_count) ? AL[k] : BL[k];
        fg_pix03[k] = (indx+4 >= fg_count) ? AL[k] : CL[k];
        fg_pix04[k] = (indx+6 >= fg_count) ? AL[k] : DL[k];
        if (indx+8 >= fg_count)
        EL[k] = AL[k];
}

// let the secondary FPGA get set up with the foreground pixel data
send_perms(OBM_A | OBM_B | OBM_C | OBM_D | OBM_E | OBM_F);

// wait for the secondary FPGA to return
send_perms(0);

// copy the background pixel data from CPU memory to FPGA on board
// memory – half to OBM A and half to OBM B
DMA_CPU(CM2OBM, AL, MAP_OBM_stripe(1,"A,B"), bg_pixel, 1, bg_count*4, 0);
wait_DMA(0);

// give the secondary FPGA control of its OBM data
send_perms(OBM_B | OBM_D | OBM_F);

read_timer(&t1);   // "distance compute time" (Figure 17) section starts here

// main unrolled computational loop – for all background pixels
nofiterations = count*(((bg_count % 4) ? bg_count+3 : bg_count)/4);
for (k = 0; k < nofiterations; k++)
{
        // i and j (image pixel row and column) index counters
        cg_count_ceil_32 (1, 0, k == 0, count-1, &j);
        cg_count_ceil_32 (j==0, 0, k == 0, 0xffffffff, &i);

        // parallel distance function calls 1-5
        distance(AL[i], fg_pix01[j], &mind01a, &mind01b);
        distance(AL[i], fg_pix02[j], &mind02a, &mind02b);
        distance(AL[i], fg_pix03[j], &mind03a, &mind03b);
        distance(AL[i], fg_pix04[j], &mind04a, &mind04b);
        distance(AL[i], EL[j], &mind05a, &mind05b);

        // find the minimum for each set (a,b) of the five distance calculations
        // (combine the results of the parallel calculations)
        mindAa = min5(mind01a, mind02a, mind03a, mind04a, mind05a);
        mindAb = min5(mind01b, mind02b, mind03b, mind04b, mind05b);

        // keep running minimum distances for each pair (a,b)
        cg_accum_imin_32(mindAa, 1, 100000, j == 0, &min_da);
        cg_accum_imin_32(mindAb, 1, 100000, j == 0, &min_db);

        // take the square root of this iteration's pair (a,b) estimated distances,
        // combine, and store
        comb_32to64_flt_flt(sqrt(min_da), sqrt(min_db), &CL[i]);
}
```

```
        // sync primary and secondary FPGA and get all OBM permissions back
        send_perms(0);

        read_timer(&t2);    // "DMA data out" (Figure 17) section starts here

        // transfer out result arrays
        DMA_CPU(OBM2CM, CL, MAP_OBM_stripe(1,"C,D"), bg_distance, 1, bg_count*4, 0);
        wait_DMA(0);

        read_timer(&t3);    // end timer for "DMA data out"

        *tm1 = t1 - t0;    // store "DMA data in" time
        *tm2 = t2 - t1;    // store "distance compute time"
        *tm3 = t3 - t2;    // store "DMA data out" time
}
```

```
int32_t min5(int32_t a, int32_t b, int32_t c, int32_t d, int32_t e)
{
        int32_t ab, cd, cde;

        ab = (a < b) ? a : b;
        cd = (c < d) ? c : d;
        cde = (cd < e) ? cd : e;

        return (ab < cde) ? ab : cde;
}
```

We added another inlined function, min5(), a function to find the minimum of five integers. We needed this because we have five parallel distance calculations that return one-fifth of the distance calculations. min5() is used to combine these results.

Note that the FPGA code has to do more work now – it needs to bring in all of the foreground pixels first and divide them up among the 5 arrays before bringing in the background pixels. There is also some code that controls which of the two FPGAs has access to which OBM banks.

The MAP® compiler shows that the unrolled loop was successfully pipelined and its pipeline depth is 90:

```
##################        INNER LOOP SUMMARY      ####################
loop on line 57:
    clocks per iteration:    1
    pipeline depth:         10

loop on line 86:
    clocks per iteration:    1
    pipeline depth:         90
#########################################################################
```

The Xilinx place and route tools show that the design fits in and meets timing specifications:

```
###############        PLACE AND ROUTE SUMMARY    ####################
```

```
  Number of Slice Flip Flops:      26,479 out of  67,584   39%
  Number of 4 input LUTs:          17,819 out of  67,584   26%
  Number of occupied Slices:       17,558 out of  33,792   51%
  Number of Block RAMs:               128 out of     144   88%
  Number of MULT18X18s:               140 out of     144   97%
  freq = 100.1 MHz
########################################################################
```

Let's have a look at the code performance executed with 32768 foreground pixels:

```
-----------------------------------------------------------
CPU time (overall)      : 2.029390 seconds
CPU time (compute only) : 2.017544 seconds
-----------------------------------------------------------
FPGA time          : 1.880733 seconds (188073328 clock cycles)
  - data transfer  : 0.001569 seconds (156891 clock cycles)
    - in           : 0.000855 seconds (85492 clock cycles)
    - out          : 0.000714 seconds (71399 clock cycles)
  - COMPUTE        : 1.879164 seconds (187916437 clock cycles)
-----------------------------------------------------------
```

This time we spent 187916437 clock cycles to perform actual calculations, which is about 1/10 of what our original code produced. The overall compute time is now 2.02 seconds compared to 18.98 seconds in the first FPGA implementation. This gives us 17x speedup compared to only about 1.8x produced by our first implementation. Figure 22 shows the complete performance results for this implementation.

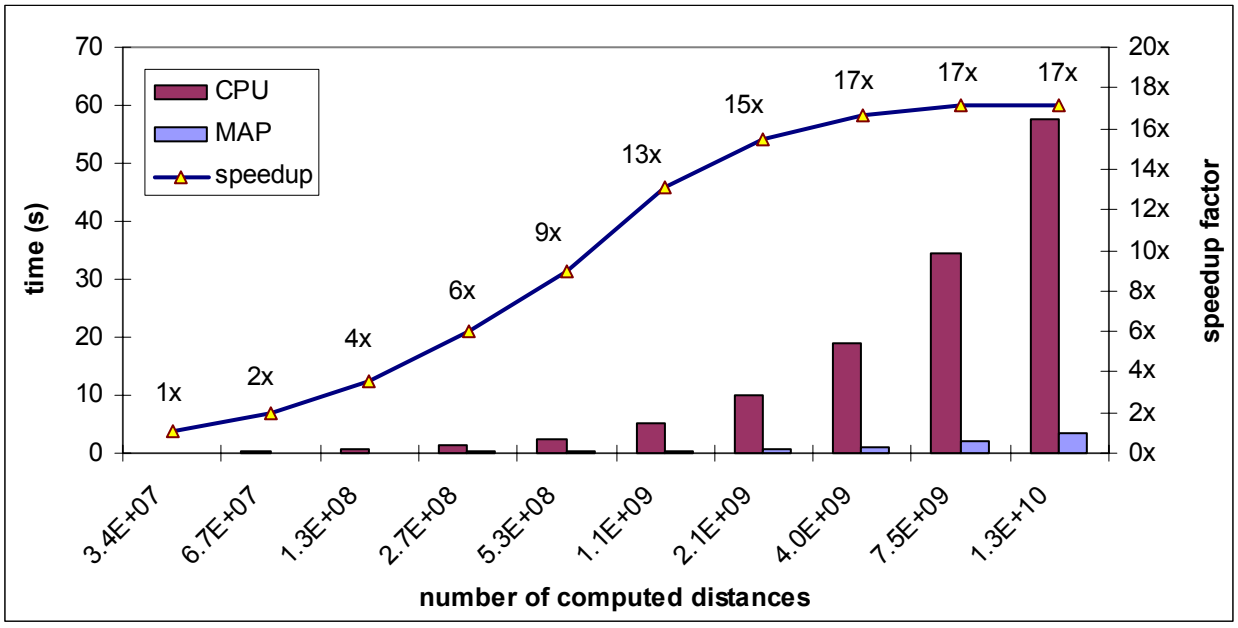

Figure 22 - CPU vs. FPGA Performance (2)

Is our 17x performance improvement for the image distance transform algorithm the best possible speedup? The answer is "no". In the near term, perhaps there is some clever algorithm

operation portioning or data organization that could yield even better performance. In the longer term, the growth in FPGA resources and clock rates guarantees better application performance.

6 Summary

In this white paper, we introduced how to get started with application programming in C on reconfigurable systems, discussed metrics that qualitatively indicate the potential value of porting a given application to a reconfigurable system, and then walked through three example iterations of algorithm development on an SRC MAPstation. We discussed reconfigurable system architectures and various approaches for a programmer to gain application acceleration, and in doing so, cautioned the programmer that reconfigurable systems for scientific applications are a relatively new development in computer architecture. And like all new developments in computer architecture, reconfigurable systems require that an application programmer learn a new set of best practices and rules of thumb ("application programming").

The FPGA portion of today's reconfigurable systems does not have the nearly infinite resources that programmers currently enjoy on CPU-only systems. Working with relatively limited resources requires some work and cleverness on the part of the programmer, but even so, we have developed performance improvements of 3x to 40x on reconfigurable systems for several scientific applications with real world data sets. Remember the limited resources of the computers of even five years ago? Contrast that with the resources of today's CPU-only systems and keep in mind that the technology growth curve for FPGAs exceeds that predicted by Moore's Law for CPUs. We are *starting* at 3x to 40x performance improvements today. We do not dare to predict typical performance improvements realizable from reconfigurable systems over the next five years for fear of being thought short-sighted by scientific application programmers in 2010.

7 Acknowledgements

This work was performed at the National Center for Supercomputing Applications and funded by the National Science Foundation (NSF) grant SCI 05-25308.

We would also like to thank the following people for their guidance, support and encouragement:

Jon Huppenthal, CEO, SRC Computers, Inc.
Rob Pennington, CTO, National Center for Supercomputing Applications, UIUC
Dan Poznanovic, VP Software Development, SRC Computers, Inc.
Dave Raila, Sr. Research Programmer, Department of Computer Science, UIUC
Craig Steffen, Sr. Research Scientist, National Center for Supercomputing Applications, UIUC