ARMY RESEARCH LABORATORY

Mstack: A Lightweight Cross-Platform Benchmark for Evaluating Co-processing Technologies

by Mark Pellegrini and Daniel M. Pressel

ARL-MR-0683 **December 2007**

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-MR-0683 **December 2007**

Mstack: A Lightweight Cross-Platform Benchmark for Evaluating Co-processing Technologies

Mark Pellegrini and Daniel M. Pressel
Computational and Information Sciences Directorate, ARL

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
December 2007	Final	June 2006–August 2006

4. TITLE AND SUBTITLE
Mstack: A Lightweight Cross-Platform Benchmark for Evaluating Co-processing Technologies

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Mark Pellegrini and Daniel M. Pressel

5d. PROJECT NUMBER
8UH3CC

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRD-ARL-CI-HC
Aberdeen Proving Ground, MD 21005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-MR-0683

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT
Co-processing technologies are currently increasing in performance at a rate superior to Moore's law, making porting applications to them desirable. However, with so many different competing co-processing technologies, the need for a simple, lightweight cross-platform benchmark has become apparent. To fill this need, we have devised Mstack, a lightweight benchmark designed to be run on a number of different classes of co-processors.

We have implemented or are in the process of implementing Mstack on a variety of different co-processor architectures: on field programmable gate arrays (FPGAs) using Mitrion-C and Dime C, graphical processing units (GPUs) using Compute Unified Device Architecture (CUDA), and Cyclops64.

15. SUBJECT TERMS
FPGA, field programmable gate array, high performance computing, HPC, GPGPU, benchmark, GPU

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Daniel M. Pressel
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL	48	19b. TELEPHONE NUMBER (*Include area code*) 410-278-9151

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

We would like to gratefully acknowledge the following people and groups for their assistance. Without their help, this work would not have been possible:

- Mitrionics and their staff, and Jace Mogill and James Maltby in particular.
- The U.S. Navy Research Laboratory, Washington, DC.
- The Department of Defense High Performance Computing Modernization Program (HPCMP) office.
- The Computer Architecture and Parallel Systems Laboratory (CAPSL) at the University of Delaware.

INTENTIONALLY LEFT BLANK.

1. Introduction

Co-processing technologies are currently increasing in performance at a rate superior to Moore's law,* making porting applications to them desirable. However, with so many different competing co-processing technologies, the need for a simple, lightweight cross-platform benchmark has become apparent. To fill this need, we have devised Mstack, a lightweight benchmark designed to be run on a number of different classes of co-processors.

The Mstack benchmark is based on the concept of a median stack. This concept is used in the oil industry for processing particularly noisy seismic data (e.g., drag data from swampy terrain). The primary nature of this process is to perform a sort on a small data set (for our purposes, 5–128 elements in size), so that the median value of the data set may be found. While this would not appear to be a particularly challenging problem, there would normally be a nearly infinite number of such data sets to process. For simplicity, we have truncated this to roughly 1-million data sets. So that we could be certain we were measuring the performance of the central processing unit (CPU)/co-processor, memory system, and associated interconnections, we used three very simple dummy test cases created "on the fly" at run time (this avoids the significant amount of time associated with random number generators and/or disk/tape input/output [I/O]).

We have implemented or are in the process of implementing Mstack on a variety of different co-processor architectures: on field programmable gate arrays (FPGAs) with Mitrion-C and DIME-C, NVIDIA's† newer graphical processing units (GPUs) which use Compute Unified Device Architecture (CUDA), and the Cyclops64‡ architecture. For our target FPGA platforms, we have intentionally selected a number of high level C-to-RTL (register transfer level) languages, which are compiled to VHSIC (very high speed integrated circuits) hardware description language (VHDL) before being mapped on to the FPGA. Cyclops64 is a massively multicore architecture being developed by IBM. The economics of the video gaming industry have made GPUs a feasible high performance computing platform. We believe our benchmark is an appropriate tool for evaluating all these technologies, as well as a number of other co-processing platforms.

In section 2, we describe the trend toward parallelism and co-processing technologies; in section 3, we discuss the design considerations and implementation choices we have made in devising Mstack; in section 4, we review the state of reconfigurable computing, i.e., computers that use FPGAs as co-processor technologies, and an array of new high-level methods of programming

*"Moore's law is the empirical observation made in 1965 that the number of transistors on an integrated circuit for minimum component cost doubles every 24 months. It is attributed to Gordon E. Moore, a co-founder of Intel." – Wikipedia, Moore's Law.

†NVIDIA, which is not an acronym, is a registered trademark of NVIDIA Corporation.

‡Cyclops64 is a trademark of International Business Machines (IBM) Corp.

them; in section 5, we report our results; in section 6, we analyze our results and draw conclusions; in section 7, we describe our planned future work in this area; appendix A contains the Mstack C reference implementation; appendix B contains the Mstack FORTRAN* reference implementation; appendix C contains technical discussions of co-processing platforms we plan to port Mstack to in the near future (in furtherance of the future work described in section 7).

2. Background

A co-processor is a device that does computation in place of the CPU. Co-processing technologies are particularly "hot" right now because their performance is increasing faster than CPU performance (see figures 1 and C-1). This trend makes porting applications to co-processors particularly desirable. Porting the application to a co-processor results in performance gains from hardware upgrades that outstrip even Moore's law.

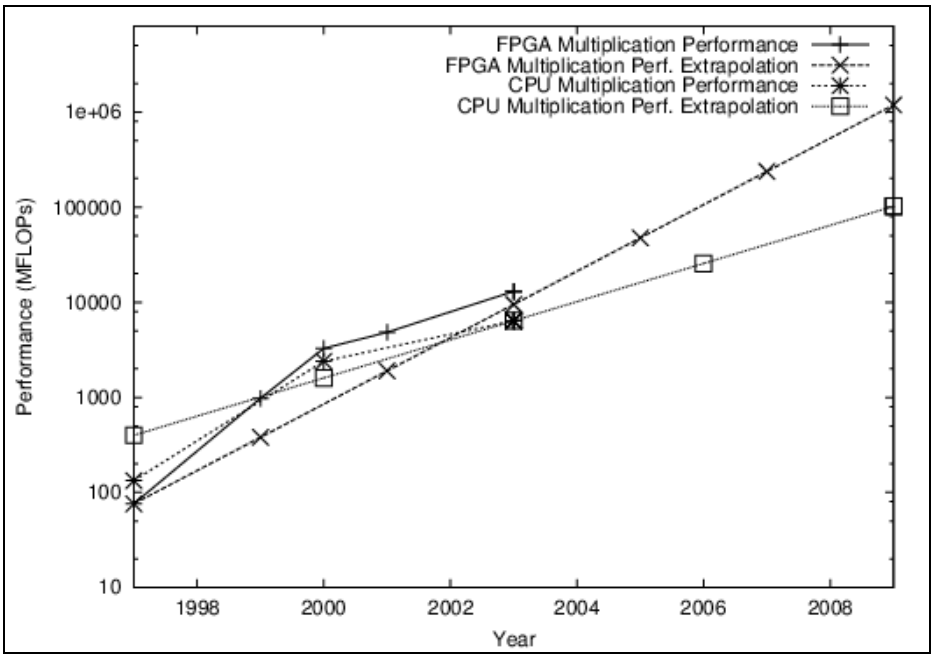

Figure 1. Comparison of CPU and FPGA single precision floating point multiplication performance (Underwood, 2004).

However, at the same time, there has been a great diversification in co-processor technology. Presently, many different kinds of co-processing technologies are vying for dominance:

- GPGPU – General purpose computation on graphical processing units,
- Reconfigurable computing – The use of application-specific logic run on FPGA co-processors,

*Formula Translator.

- Multicore and many-core architectures – High Performance Computing systems with many relatively simple cores. Examples in this class include the Cell Broadband Engine,* Sun UltraSPARC T1† (Niagara), and IBM Cyclops64.

Each of these classes has its advantages and disadvantages. A number of benchmarks have been created to test FPGAs (McCarty et al., 1993; Govindaraju et al., 2005) and GPUs (Buck et al., 2004; Kumar et al., 2000). However, these benchmarks are limited to coprocessors within a given class. To our knowledge, no inter-class co-processor benchmark has been created.‡ The goal of the Mstack benchmark is to evaluate the co-processing technologies on an apples-to-apples basis to provide a valid comparison between very different specialized devices. Ultimately, Mstack's purpose is enable users to evaluate co-processors' potential for novel concepts in high performance computing—concepts that might appear as options in future high performance computing (HPC) systems.

3. The Mstack Benchmark

We have devised a small, easy-to-compile benchmark based on determining the median value in an unsorted array. Determining the median requires sorting the array, and picking the value in the center element in the array. To sort the array we chose to use the bubblesort algorithm because of its simplicity and that for small arrays it may be nearly optimal performance-wise.

3.1 Design Considerations

The Mstack benchmark was designed under the "keep it simple stupid" philosophy. It was designed to be practical to run on CPUs as well as on a wide variety of co-processors. Some of the design requirements and constraints that went into developing the benchmark were

1. The run time must be long enough that it makes sense to accelerate the application. At the bare minimum, this means run times with a length of minutes to hours.
2. The run time must be short enough that it is reasonable to do at least some runs with a simulator or hardware emulator. This will normally favor a run time of 1 s or less, but a few seconds of simulated time should be doable in less than 12 hr of simulation time.
3. It should fit in main memory but require vastly more space than is likely to be available in the caches/dedicated local memory.

*Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

†UltraSPARC T1 is a trademark of Sun Microsystems.

‡We are neglecting all-purpose benchmarks and micro-benchmarks such as fast Fourier transform and matrix multiply which may not be suitable for highly specialized/optimized architectures devices such as GPUs and FPGAs.

4. The amount of input/output (I/O) should be negligible. After all, we are attempting to measure computational performance.

5. Although it might have been desirable to have an inner loop that can be parallelized and vectorized, for many of the platforms undergoing consideration, what matters is having middle and/or outer loops that parallelize.

6. An embarrassingly parallel algorithm was selected, since this minimizes the effort associated with parallelization.

7. We needed a floating point algorithm, but for simplicity's sake, it was desirable to find as simple an algorithm as possible. That is to say, not a complete code with 100 or more boundary condition routines to worry about, but many simple benchmarks such as matrix multiply and FFTs that have already been implemented and are well known in the field.

Keeping these constraints in mind and drawing on Mr. Pressel's earlier experience in processing seismic data for the oil industry, we constructed a benchmark that extracts the key components of a median stack. In seismic data processing, stacking is a method of averaging several values that nominally correspond to the same subsurface location. Ordinarily, this is done with the arithmetic mean. However, in situations when the data are noisy (for example, drag data recorded in swampy locations), the presence of outliers can heavily skew the results. A commonly used alternative for noisy data is the median stack, where the data values to be averaged are first sorted so that it is possible to determine what the median value is. The issues involved with this process are

1. The number of values to be sorted is usually small (never greater than 128, and possibly less than 24). Therefore, we decided to benchmark data sets where the number of channels are 5, 50, 75, and 128. The number of channels determines the number of values to be sorted and is physically analogous to the number of sensors recording data after each explosion.

2. The number of data sets to be averaged would normally be nearly infinite. For purposes of this benchmark, it was decided to use 1 million data sets, although some of them are not processed since they would be incomplete (representing the seismic boat turning around). When one is using 32-bit floating point data, this should comfortably fit into 1 GB of main memory (depending on the implementation, the required amount of main memory is in the range of 500 to 600 MB). Note that in order to be faithful to the origin of this benchmark, it is important that all the data be written to main memory before the start of the processing.

3. Since we lacked ready access to actual data, three data sets were constructed for benchmarking purposes. The first sets all the values to a constant value (1.0 in this case). The second sets the values of each channel to the channel number, so the values go from 1.0 to number_of_channels (in floating point). The third data set assigns numbers to the channels in reverse order and is the only data set that is not already sorted. These three data sets were selected for simplicity and since they are known to be inefficient for one or more of the commonly used sorting algorithms.

4. We selected the bubblesort for use as the sorting algorithm. For smaller numbers of channels, other sorts are unlikely to show their asymptotic performance, and the bubblesort will likely be faster because of its simplicity. This simplicity will also have benefits when the benchmark is running on single-instruction-multiple-data (SIMD) arrays of processors with small amounts of instruction cache per processor. Similarly, bubblesort should make it easier to implement on FPGAs. While the reference implementations do not explicitly show the use of predicated operations, it was not difficult to produce a C implementation that uses the conditional operator and therefore lends itself to being implemented with predicated operations. This should improve the performance on the GPUs. Additionally, note that the performance of the bubblesort is nearly data independent.
5. Both FORTRAN 77 and C versions of the bubblesort were implemented, including OpenMP directives for the parallelization of the outer loop on shared memory platforms.
6. A minimal amount of output is produced. This is helpful for debugging purposes and to disable overly aggressive compiler optimizations.

In order to avoid limitations on the stack size and stack frame size for many platforms, the largest array (roughly 500 MB in size) is static allocated onto the heap (**save** statement in FORTRAN, **static** qualifier in the declaration for C). In order to maximize the similarity between the FORTRAN and C code, the decision was made not to use pointers in either version. This means that no dynamic memory allocation was used in any of the versions and that array syntax is used throughout the C versions of the benchmark.

3.2 Mstack Implementation

The benchmark incorporates a bubblesort. However, for comparison purposes, other sorting algorithms can be used if it is felt that they are better suited to a particular platform. In these cases, it is important to specify which sort has been used, and it is preferred that some amount of justification be included.

The only input that the benchmark takes is at the beginning of the run, when it prompts the user for the number of channels. The purpose of this is twofold: it allows the user to control how long the benchmark runs, in effect, replicating the computation classes used in other benchmarks, such as the NASA Advanced Supercomputing (NAS) Parallel Benchmark Suite. Secondly, it imposes a compile-time knowledge limitation on the compiler. Thus, the compiler cannot optimize the sorting work at the heart of the benchmark.

The original C and FORTRAN reference implementations are given in appendices A and B, respectively. We have also produced alternate implementations of Mstack in C and FORTRAN which use different optimization schemes to attempt to achieve better performance on a range of architectures. Mstack2 and Mstack3 differ from the reference implementation by using the

ternary operator instead of an "if" statement. This may make a difference with some compilers, allowing them to produce more efficient assembly code using predicated instructions. Mstackv is a vectorized version of the Mstack. This vectorization allows pipelining or parallelization of the innermost loop. The code for the different variations of the bubblesort is given in section 3.3.

For each of these four versions (the original reference implementation, mstack2, mstack3, and mstackv), we also applied an optimization to the inner loop whereby it would sort to the length of the vector minus the iteration. The reference implementation sorted through the entire vector on every iteration, thus performing N^2 comparisons and possible swaps. The optimized versions perform $1/2 \times N^2$ comparisons and possible swaps. We consider all versions of the Mstack benchmark to be acceptable and equally valid for the purposes of testing hardware. We have tested each of the four versions (mstack, mstack2, mstack3, and mstackv) and their optimized equivalents (mstacko, mstack2o, mstack3o, and mstackvo) in C and FORTRAN for 50 channels. Our results are given in section 5.

3.3 Table of Mstack Bubblesort Variations

The bubblesort functions for each of the non-optimized versions of Mstack are shown in table 1.

Table 1. Bubblesort implementations for the various versions of Mstack.

```
/* Mstack */
/* Perform a bubblesort */
for (k1=1; k1 <= numchn; k1++)
{
  for (k2=1; k2 <= numchn-1; k2++)
  {
    if (scratch[k2] > scratch[k2+1])
    {
      temp = scratch[k2];
      scratch[k2]=scratch[k2+1];
      scratch[k2+1]=temp;
    };
  };
};
```

```
/* Mstack2 */
/* Perform a bubblesort */

for (k1=1; k1 <= numchn; k1++)
{
  for (k2=1; k2 <= numchn-1; k2++)
  {
    temp1 = scratch[k2];
    temp2 = scratch[k2+1];
    temp3 = temp1 > temp2? ((temp4=temp1),temp2):((temp4=temp2),temp1);
    scratch[k2]=temp3;
    scratch[k2+1]=temp4;
  };
};
```

Table 1. Bubblesort implementations for the various versions of Mstack (continued).

```
/* Mstack3 */
/* Perform a bubblesort */
for (k1=1; k1 <= numchn; k1++)
{
  temp1 = scratch[1];
  for (k2=1; k2 <= numchn-1; k2++)
  {
    temp2 = scratch[k2+1];
    temp3 = temp1 > temp2? ((temp4=temp1,temp2):((temp4=temp2),temp1);
    scratch[k2]=temp3;
    temp1 = temp4;
  };
  scratch[numchn]=temp4;
};
```

```
/* Mstackv */
/* Perform a bubblesort */
for (k1=1; k1 <= numchn; k1++)
{
  for (k2=1; k2 <= numchn-1; k2++)
    for (l=1; l <= 1001 - numchn; l++)
    {
      temp1 = traces2[k2][l];
      temp2 = traces2[k2+1][l];
      if (temp1 > temp2)
         {
          traces2[k2][l]=temp2;
          traces2[k2+1][l]=temp1;
          };

    };
};
```

4. Reconfigurable Computing and the Mitrion-C Language

4.1 Reconfigurable Computing

One recent trend in HPC architecture is to integrate FPGAs into HPC systems as co-processors. These systems are known as “reconfigurable computers” because the logic on the FPGA can be reconfigured for each application run on the system or even for different sections of a given application. In order to get an acceptable level of performance, the FPGA must have high bandwidth, low latency access to the system memory. This requires a high degree of cooperation between the system, FPGA, and bus vendors, which did not exist until recently. According to Michael D’Amour, chief executive officer of DRC* Computer Corporation, “It took 18 months for DRC to convince Advanced Micro Devices (AMD) to open [their Hypertransport bus] up…

*DRC is not an acronym.

When we first walked into AMD, they called us 'the socket stealers'. Now they call us their partners." AMD's decision to open their Hypertransport bus protocol to third parties has made Hypertransport the enabling technology in reconfigurable computing (D'Amour, 2007). However, a large number of commercially available plug-in FGPAs use the much slower PCI/PCI-X/PCIe bus, and these busses are the only option for systems using non-AMD processors or non-Xilinx FPGAs.

Previous work in this area indicates that migrating certain classes of applications to FPGAs has the potential for tremendous acceleration. (Pressel, 2007; Zhang et al., 2007). However, recognizing that hardware description languages (HDLs) such as VHDL and Verilog* are difficult and time consuming to program (akin to programming a CPU in assembly language), a number of "C-to" (alternatively, "C-to-RTL") languages have been developed. These are C-like languages that are compiled to HDL and then compiled again to an FPGA bit mapping. These C-like languages include

- Mitrion†-C – commercially available for SGI‡ RC100 and Cray XD1
- DIME-C – a language for Nallatech FPGAs only
- Handel§-C – "Basically a small subset of C extended using a few constructs for configuring the hardware device" (Peter, 2007)
- Impulse**-C – Commerical. Supports several FPGAs (Altera Cyclone, Altera Stratix, Xilinx Spartan, Xilinx Virtex) on many platforms

The C programming language is, by its very nature, targeted for CPU architectures. This means that C programs generally do not map to hardware easily or efficiently. To give one very common example, consider the following line of C code:

```
a = x[y];
```
(1)

where y is not known at compile time. On a CPU system, this operation is trivial to implement. However, because y is not known at compile time, the hardware compiler is unable to optimize this instruction. Thus, implementing this in hardware requires that x is stored in its entirety in a programmable logic array (PLA). If x is large, this can consume an enormous number of transistors. Accordingly, the Mitrion-C manual describes vectors (Mitrion's equivalent to arrays in C) by saying, "You will soon notice that using even moderately large vectors can use all the available resources. A vector may be indexed but preferably only by compile-time constant values. Accessing data-dependent indices is very costly on silicon surface and should be

*Verilog is a trademark of Cadence Design System, Inc.

†Mitrion is a trademark of Mitrionics AB.

‡SGI is a trademark of Silicon Graphics, Inc.

§Handel is a trademark of Celoxia, Inc.

**Impulse is a trademark of Impulse Accelerated.

avoided. If data-dependent access to a vector's elements is required, you should consider if an internal RAM-memory would not be more suitable" (Möhl).

Different "C-to" languages take different approaches to programmability and efficiency. DIME-C and Impulse-C are subsets of the C-language, making them particularly easy to program and debug but at the cost of being inefficient to map to an FPGA. In contrast, Mitrion-C resembles C only superficially. It uses much of the same syntax, but conceptually, it is vastly different. As a result, Mitrion-C is more difficult to program but has the potential to map to the FPGA very efficiently.

4.2 The Mitrion-C Language

Mitrion-C's resemblance to C is only skin deep. Beyond some basic syntactical similarities, it is a totally different language.

Mitrion-C is a single assignment, fully implicitly parallel language. This prevents the programmer from creating false dependencies and allows the compiler to create a data flow model of the program in hardware. This, in turn, exposes all the thread-level parallelism in the program, allowing the compiler to generate an efficient parallelized hardware implementation. Mitrion's software development kit (SDK) includes a visual debugging feature that allows the developer to see values flowing through a data flow representation of the program.

Explicit data typing is, in most cases, optional. However, all declarations must specify a bit width. Mitrion-C is capable of handling arbitrarily large integers and floating point variables. Floating point declarations must specify both the mantissa and exponent bit widths. An Institute of Electrical & Electronics Engineers (IEEE) 754 single precision floating point variable is declared as

```
float:24.8 foo;
```

An IEEE 754 double precision floating point values is declared as

```
float:53.11 bar;
```

Mitrion-C includes three looping constructs: for, foreach, and while. The foreach loop is used to signify that the number of iterations the loop will execute is known at compile time and that there are no loop-carried dependencies. The for loop indicates that the number of iterations is known at compile time and that there are loop-carried dependencies. The while loop is used to indicate that the number of iterations is not known at compile time. However, the language requires the program to make a guess.

Mitrion-C has four collective data types:

- List – An unindexed array whose length is known at compile time. Element-wise operations occur sequentially.

- Vector – An indexed array whose length is known at compile time. Element-wise operations occur in parallel.
- Stream – An unindexed array whose length is not known at compile time. Element-wise operations occur sequentially.
- Tuple – A "bag" of various data types analogous to the tuple found in many programming languages (Python, etc.).

Lists are analogous to elements traveling through a hardware pipeline. Lists, even very large ones, tend to use very few FPGA resources beyond the logic already specified by the program. Vectors are buffered memory allocations analogous to programmable logic arrays. Vectors are easy to program with but very costly in silicon.

Everything in Mitrion returns a value (functions, loops, if/else statements, etc.). Given that Mitrion-C will be instantiated in hardware, this is entirely sensible. It is illogical to instantiate hardware that does not contribute to the computation of the program.

A proper Mitrion-C program includes a Mitrion-C file (that uses the .mitc extension) and a C program that runs on the host. The Mitrion-C file is compiled by the mitrion compiler into a VHDL program. This VHDL program is, in turn, compiled into a bit mapping that is loaded onto the FPGA at run time.

Host-FPGA communication can be done in one of two ways with the use of vendor-supplied communication libraries or Mithal. Mithal is a Mitrioncs-supplied communication library that acts as a wrapper for a number of vendor-supplied host-FPGA communication libraries. Mithal includes the features common to all those libraries. This allows Mitrion-C programs to be portable to any architecture whose host-FPGA communication libraries are supported by Mithal. On the host side, the communication is instantiated as a DMA. The host-FPGA interaction is shown in figure 2.

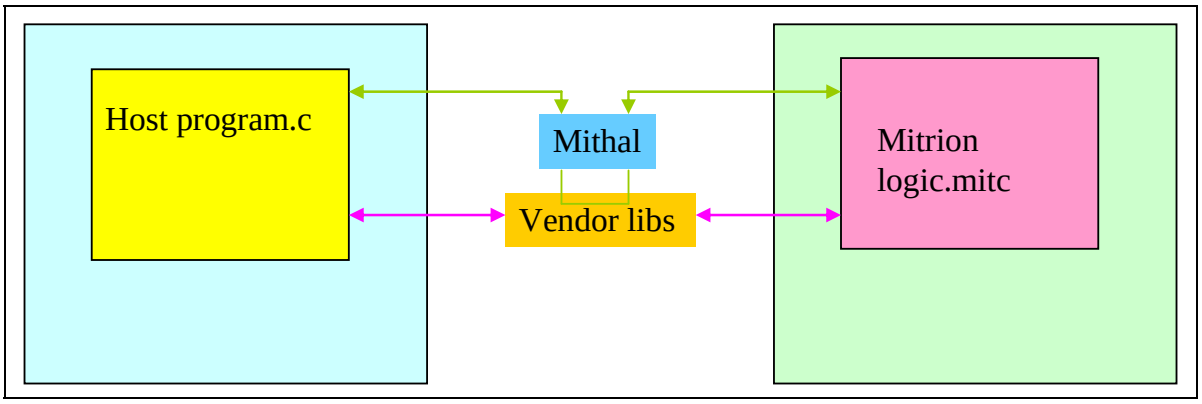

Figure 2. A logical view of the Mitrion-C host/FPGA interaction. The host communicates to the FPGA using (1) vendor-supplied libraries (shown with pink arrows), or (2) Mithal, a Mitrionics-supplied library that runs on top of the system vendor's FPGA communication libraries (shown with green arrows). (By allowing the program to avoid system-specific [or vendor-specific] communication libraries, Mithal allows Mitrion programs to be cross-platform compatible.)

Consider the following code from the host-side C file:

```
printf("Allocating FPGA... ");
my_fpga = mitrion_fpga_allocate("");
if(my_fpga == NULL) {
   fprintf(stderr, "Couldn't find the FPGA!\n");
   return 22;
}
printf("Done\n");

printf("Create processor... ");
my_proc = mitrion_processor_create(argv[1]);
if(my_proc == NULL) {
  fprintf(stderr, "Couldn't create the MVP!\n");
  return 23;
}
printf(" Done\n");

mem_a = mitrion_processor_reg_buffer(my_proc, "mem_a", NULL,
                                                sizeof(int),
                                                WRITE_DATA);

printf("Invoking asynchronous run...\n");
mitrion_processor_run(my_proc);

printf("Waiting for processor...\n");
mitrion_processor_wait(my_proc);
printf("Done\n");
```

The code just given declares the FPGA, the Mitrion virtual processor (MVP), and mem_a, the direct memory access (DMA) memory array. It then invokes an asynchronous run, letting the processor begin to run the bit mapping of the Mitrion-C file. Meanwhile, the host side waits for the FPGA to finish.

The mitrion_fpga_allocate, mitrion_processor_create, mitrion_processor_reg_buffer, mitrion_processor_run, and mitrion_processor_wait functions form the core of the Mithal application program interface (API). The mitrion_processor_reg_buffer is the function that actually allocates the DMA memory—the mem_a array.

The Mitrion-C SDK includes an FPGA simulator for Mitrion-C files. Given the long Mitrion-to-VHDL-to-bit mapping compilation time (about 15 minutes on our target platform for the most trivial Mitrion-C program), the utility of this tool for code development, testing, and debugging cannot be overstated. Figure 3 contains a diagram of this feature.

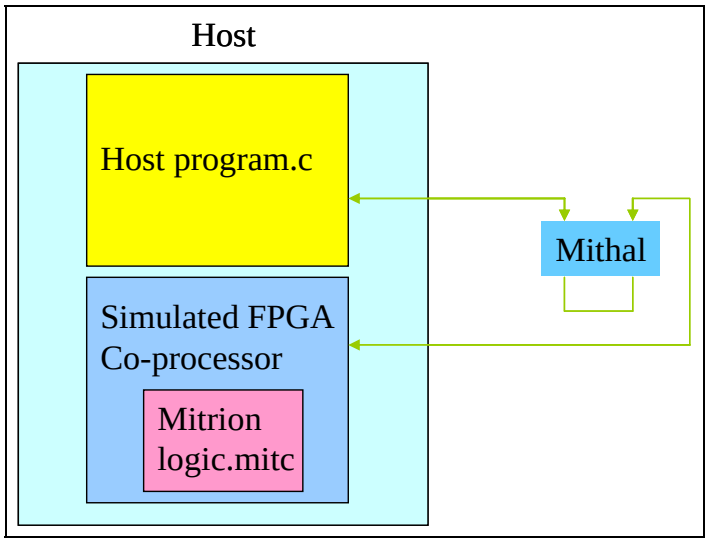

Figure 3. Mitrion-C's SDK includes FPGA simulation capabilities, allowing developers to test Mitrion-C programs on systems that are not FPGA equipped.

Memory accesses in Mitrion are done with instance tokens. An instance token is both a pointer to a particular memory location and a representation of the state of that memory at a given point in program execution. Each memory operation consumes a memory token and returns a new one to be used in the next memory operation. Thus, a name_number naming scheme is prudent. This pattern is illustrated in the following code segment:

```
1. float:53.11 tmp;
2. p0 = _memcreate (mem float:53.11 [100] p_last);
3. p1 = _memwrite(p0, 3, 1.7);
4. p2 = _memwrite(p1, 5, 9.4);
5. (tmp, p_last) = _memread(p2, 9);
```

Line 1 declares a temporary double precision floating point variable to be used later. Line 2 allocates a block random access memory (RAM) segment of 100 double-precision elements; p_last, which is declared here, indicates to the compiler the last memory token that can be expected to access this memory. When the program reaches the point at which p_last is assigned, the block RAM can be de-allocated. Line 3 writes 1.7 to the third element of the array, and line 4 writes 9.4 to the fifth element in the array. Line 5 reads the values stored in the ninth element in the array, and stores it into tmp (which was declared on line 1). Notice that each memory operation consumes the previous instance token and returns a new one. The compiler uses this to enforce sequential consistency in memory operations; the memory operation on line 4 will not occur until after line 3 has compiled, line 5 will not occur until after line 4 has completed, etc.

5. Results

5.1 Baseline Results

For our baseline testing, we used three ARL Major Shared Resource Center (MSRC) unclassified clusters: JVN, MJM,* and Powell. Their various hardware characteristics are presented in (ARL MSRC System Overviews, 2007).

For each of the 16 versions, we tested them with 50 channels. Our results are shown in table 2.

Table 2. Relative performance of various versions of Mstack on three ARL MSRC clusters.

Version	Powell	JVN	MJM
Mstack.f	1	1	1
Mstack2.f	0.743	0.768	1.349
Mstack3.f	0.870	0.890	0.756
Mstackv.f	0.960	0.840	1.298
Mstacko.f	1.728	1.693	2.037
Mstack2o.f	1.382	1.325	2.146
Mstack3o.f	1.653	1.475	1.431
Mstackvo.f	1.732	1.544	2.810
Mstack.c	1.133	0.809	0.997
Mstack2.c	0.775	0.894	0.851
Mstack3.c	0.926	1.131	0.909
Mstackv.c	1.090	1.647	1.309
Mstacko.c	1.995	1.831	2.009
Mstack2o.c	1.281	1.368	1.385
Mstack3o.c	1.563	1.562	1.392
Mstackvo.c	1.854	2.027	1.764

The values for each of the different systems are the acceleration relative to the run time of the FORTRAN reference implementation on that given system. The run times of these referenced FORTRAN implementations are 60.162 s for Powell, 50.686 s for JVN, and 29.49 s for MJM.

There does not appear to be a clear pattern as to which language produces faster code. Depending on the version, the system, and the compiler, sometimes one was faster and sometimes the other was faster. In many cases, the differences between the C and FORTRAN versions were insignificant. The optimized versions, which run half as many computations, outperformed the non-optimized versions in every case. Unfortunately, this optimization might be difficult to use on some of the novel platforms that we plan to investigate. In both FORTRAN and C, the mstack2 and mstack3 implementations generally did worse than the reference implementation. There was no clear pattern between the performance of mstackv when compared to the reference implementation; however, in many cases, the differences in performance were negligible.

* JVN and MJM are not acronyms.

It is expected that mstack2.c, mstack3.c, or their optimized counterparts will make the best coding base when one is porting to NVIDIA GPUs via CUDA (NVIDIA, Inc., August 2007). For future work with FPGAs, it appears as though mstackv.c or mstackvo.c will make the best coding base.

5.2 Mitrion-C Results

For our Mitrion-C FPGA testing, we used Kamala, the U.S. Navy Research Laboratory's Cray* XD1. The Cray XD1 system architecture is described in detail in S.2429.131, the Cray XD1 System Overview. The Cray XD1 base system (a single chassis) consists of 31 total processors: 12 64-bit x86 AMD Opteron processors, 6 or 12 RapidArray* processors, zero or six FPGAs to act as co-processors for the Opterons, and one management processor to run monitoring and control the chassis health. Multiple chassis can be connected in customizable topologies. The 12 Opteron processors per chassis are connected as two six-way or three four-way symmetric multiprocessors. Each SMP runs an instance of a Cray-customized version of Linux based on SUSE.† Each instance of Linux (along with all the hardware that it controls) is referred to as a node. The RapidArray processors provide access to the RapidArray interconnection through which interprocessor communication within a chassis occurs. The RapidArray interconnection is "a 48-GB-per-second nonblocking embedded switch fabric, with optional expansion fabric for 96-GB-per-second capability" (Cray, Inc., 2006). The FGPA models used vary from system to system. Kamala uses the Xilinx Virtex II Pro XC2VP50. The XC2VP50 has 53,136 logic cells and 4.07 megabytes of block RAM (Xilinx, Inc., 2007).

The XD1 has four 64-bit buses for data transfer between the host program and the FPGA, but Mstack is a 32-bit (e.g., single-precision) benchmark. Rather than double-stuffing pairs of 32-bit elements into each bus transfer, we instead pad the transfer and throw away the pad. We do this because of the difficulty in unpacking double-stuffed elements using Mitrion-C. However, this does make for a slower data transfer.

Our Mitrion-C bubblesort function was vector based:

```
//mstack-specific declarations
#define WORD float:24.8
#define MAXFLOAT 0b01111111100000000000000000000000

#define MAXP1 51
#define MAX 50
#define MAXM1 49

WORD [MAX] bubblesort (WORD [MAX] scratch)
{
  //initial definition of vec
  WORD [MAXP1] vec = foreach (i in [0 .. MAX]){
    WORD v = if (i < MAX) (scratch[i]) else (MAXFLOAT);
```

*Cray and RapidArray are trademarks of Cray, Inc.

†SUSE is not an acronym.

```
  } v;

  WORD [MAX] vec2 = scratch; //arbitrary initial definition
  WORD [MAX] ret = for (k1 in <0 .. 2*MAXP1>) //replace 1 with MAXP1
  {
    WORD grtr = vec[0];

    WORD lssr = 999; //arbitrary initial definition
    vec2 = for (k2 in [1 .. MAX])
    {
      (grtr, lssr) = if (vec[k2] > grtr) (vec[k2], grtr) else (grtr, vec[k2]);
    } >< lssr;
    vec = foreach (i in [0 .. MAX]){
      WORD v = if (i < MAX) vec2[i] else MAXFLOAT;
    } v;
  } vec2;
} ret;
```

Note that the number of elements to be sorted is hard coded (e.g., #define MAXP1, MAX, and MAXM1). This requires that we (a) hard code this value to 128, regardless of the number of channels to be sorted, or (b) recompile the Mitrion-C program any time we need to simulate a number of channels greater than what the compiled version was coded for.

As previously stated, vectors (especially data-indexed vectors) consume a large amount of silicon. When the code just given is compiled, the Mitrion-C compiler complains that it is too large:

```
***ERROR: Error: Your Mitrion program uses a lot of Flip Flops
and will not be able to successfully synthesize for the target
FPGA. Try to reduce your Mitrion program to use approximately 50%
of the available Flip Flops.
```

Reducing the number of elements that the FPGA has to sort (by changing #define MAXP1, MAX, and MAXM1) reduces the silicon footprint. The vector-based version compiled successfully for, at the largest, a vector of 19 elements. Any larger than that would cause (a) a segfault in the Xilinx VHDL compiler for vectors of roughly 20 to 40 elements; (b) a too-large error from the Mitrion compiler, as shown before, for vectors of roughly 40 to 50 elements; or (c) a java out-of-memory error in the Mitrion compiler for vectors of roughly 60 elements or larger.

For five channels (the only size of the benchmark that the Mitrion-C version can run), overhead from initializing the FPGA and transferring data to it and back tends to dominate over computation. As a result, the host-only C reference significantly outperformed the FPGA implementation. The C reference version processed five channels in 3.06 s; the FPGA implementation was slower by roughly a factor of 30.

6. Conclusions

We believe Mstack has the potential to be an effective co-processor benchmark. Its relative simplicity was expected to make porting Mstack to other co-processing platforms relatively easy. Unfortunately, our experiences with Mitrion-C have shown this to be overly optimistic. Its dense, predictable memory access pattern gives co-processors the opportunity to demonstrate the suitability of their memory architecture.

Based on our experiences, we believe the Mitrion-C language is currently far too difficult to program to make it an effective option for most software development projects for high performance computing. We found it extremely difficult to develop a bubblesort function in Mitrion-C, owing primarily to the fact that (a) the language is based on the rarely used data flow (single assignment) programming model, and (b) arrays must be assigned in their entirety (e.g., you cannot assign to individual elements within the array; you have to assign the entire array). These two language characteristics make implementing any sorting algorithm inordinately difficult. It took roughly 3 weeks to devise and implement the vector-based bubblesort function shown in section 5.2. As described in section 7, we are currently working on a new implementation that eliminates vectors and uses block RAM instead, which is believed to be a more appropriate fit to this problem. However, we found it to be curious that the language does not come with library routines for any of the commonly used sorting algorithms. Virtually every major programming environment now comes with such utilities. However, we could not find any evidence that Mitrion-C includes these routines in their distribution, at least for the Cray XD1.

During initial attempts to create the simplest Mitrion-C test program possible (a program that transfers data into the FPGA and then back out again), an issue with the SDK's FPGA simulator was encountered whereby floating point values passed into the simulated FPGA are zeroed. Mitrionics staffers confirmed that this is a known simulator bug. Other than this case for that test program, the SDK's FPGA simulator performed superbly and significantly accelerated the development process by eliminating long VHDL-to-bit-mapping compilation times from the testing process.

7. Future Work

Work to develop Mstack for other co-processing platforms is on-going. We are partnering with the University of Delaware's Computer Architecture and Parallel Systems Laboratory (CAPSL) to develop Mstack on a number of other co-processing platforms. Appendix C contains a

description of some of the target platforms we are in the process of porting Mstack to. These architectures are IBM's Cyclops64, NVIDIA GPUs doing general purpose computation (also known as GPGPU) using the CUDA API, and onto FPGAs using the DIME-C language.

Other platforms to which we are considering porting Mstack include the Cray MTA2, the Cray XMT (also known as the MTA-3, or "Eldorado"), the ClearSpeed* custom processor, the Cell Broadband Engine, and to FPGAs using the Impulse-C language.

Work has started on a second Mitrion-C implementation of bubblesort that is not vector based but stores data into block RAM on the FPGA. The limited amount of block RAM makes storing all the host data on the FPGA impossible; however, the vectorized version of Mstack may be able to take advantage of this, further reducing overhead from initializing the FPGA and doing data transfer. The small size of the block RAM is the limiting factor as to how many inner loop iterations can be rolled together.

Recently, Song Park of ARL began efforts to port Mstack to DIME-C. Since this work is in its infancy, it is premature to report results for this part of this project.

*ClearSpeed is a trademark of ClearSpeed Technology Plc.

8. References

ARL MSRC Computer System Overviews. MJM - http://www.arl.hpc.mil/Systems/mjm.html; JVN - http://www.arl.hpc.mil/Systems/jvn.html; Powell http://www.arl.hpc.mil/Systems/powell.html (accessed August 2007).

Buck, I.; Fatahalian, K.; Hanrahan, P. GPUBench: Evaluating GPU Performance for Numerical and Scientific Applications. In *Proceedings of the 2004 ACM Workshop on General-Purpose Computing on Graphics Processors*, 2004.

Cray, Inc. Cray XD1 Supercomputer. http://www.cray.com/products/xd1/ (accessed 2006).

D'Amour, M. R. Standardized Reconfigurable Computing. Presentation at the University of Delaware, 28 February 2007.

Govindaraju, N. K.; Raghuvanshi, N.; Henson, M.; Tuft, D.; Manocha, D. *A Cache-Efficient Sorting Algorithm for Database and Data Mining Computations Using Graphics Processors*; UNC technical report, 2005.

Kumar, S.; Pires, L.; Ponnuswamy, S.; Nanavati, C.; Golusky, J.; Vojta, M.; Wadi, S.; Pandalai, D.; Spaanenberg, H. A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions. *Proceedings of the 2000 ACM/SIGDA Eighth International Symposium on Field Programmable Gate Arrays*, 2000.

McCarty, D.; Faria, D.; Alfke, P. PREP benchmarks for programmable logic devices. *Custom Integrated Circuits Conference, 1993, Proceedings of the IEEE, 1993*.

NVIDIA, Inc. CUDA Programming Guide. http://developer.download.NVIDIA.com /compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf (accessed August 2007).

Peter, C. Overview: Hardware Compilation and the Handel-C language, Oxford University Computing Laboratory, UK. http://web.comlab.ox.ac.uk/oucl/work/christian.peter /overview_handelc.html (accessed November 2007).

Pressel, D. M. *FPGAs and HPC*; ARL-SR-147; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, January 2007.

Möhl, S. The Mitrion-C Programming Language. Version 1.2.0-001.

Underwood, K. FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, 2004.

Xilinx, Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete data sheet; DS083 (v4.6), 5 March 2007.

Zhang, P.; Tan, G.; Gao, G. R. Implementation of the Smith-Waterman Algorithm on a Reconfigurable Supercomputing Platform. CAPSL Technical Memo 78, 16 April 2007.

INTENTIONALLY LEFT BLANK.

Appendix A. Mstack C Reference Implementation

```
/*************************************************************************/
/*                                                                       */
/* A simulated program for performing a Median Stack on Drag Seismic Data.*/
/*                                                                       */
/* This program will assume that there are a large number of data traces, */
/* with a constant number (M) channels per data trace.  While M can be any*/
/* positive integer, in general it is expected that 10 <= M <= 128, and   */
/* that the number of data traces and the number of measurements per trace*/
/* are both >> M.  For simplicity, the first M-1, and last M-1            */
/* measurements per trace will be thrown away since less than M values    */
/* exist to be stacked at those points.                                   */
/*                                                                       */
/* The median stack inherently involves performing a large number of sorts*/
/* on an almost infinite number of relatively small data sets (the set    */
/* size here will be N).  Therefore it does not make sense to use a       */
/* sophisticated sorting algorithm.  Instead, we will keep it simple by   */
/* using a bubblesort.  However, we will use OpenMP to allow multiple     */
/* sorts to be performed at once.                                         */
/*                                                                       */
/* Ordinarily the data would be read in from a file containing partially  */
/* processed data from the field.  For our purposes, the program will     */
/* populate a large two dimensional matrix (32 bit values since           */
/* measurements have limited precision, and drag data is especially       */
/* noisy) with dummy values).  In order to minimize extraneous effects,  */
/* three different dummy data sets will be created, and the program will  */
/* cycle through the three sets twice.  The sets are:                     */
/*                                                                       */
/*      1)  All values will be 1.0                                       */
/*                                                                       */
/*      2)  The values within a single channel will be the channel       */
/*          number (1 to M, where 2 <= M <= 128).                        */
/*                                                                       */
/*      3)  The values within a single channel will be                   */
/*          float(M - channel number).                                   */
/*                                                                       */
/* For simplicity, we will always use 1000 for the number of measurements */
/* per trace, and 1000 data traces.  These values were chosen so that they*/
/* will comfortably fit in 1 GB of memory, with room left over.           */
/*                                                                       */
/* NOTE:  This program does not use dynamic memory allocation.  Therefore */
/* it is possible that if the number of channels is significantly less    */
/* than 128 that one could increase the number of data traces.            */
/*                                                                       */
/* A small number of values will be output for each data set in an effort */
/* to prevent an optimizing compiler from optimizing away all of the work.*/
/*                                                                       */
/* Written by Daniel M. Pressel at ARL in July of 2007.                   */
/*                                                                       */
/* Purpose is to provide a simple reference benchmark for testing out     */
/* the ability of attached processors (e.g., FPGAs, GPGPUs, the SPEs in   */
/* the Cell processor, or Clear Speed) to accelerate floating point       */
/* applications.  This application was chosen for its simplicity.  While */
/* it may have little relevance to the military, it comes from the        */
/* problem domain of seismic prospecting for oil, which has always been an*/
/* HP/*application.                                                       */
/*                                                                       */
/*************************************************************************/
```

```
#include <stdio.h>
main (argc,argv)
int argc;
char **argv;
{
 static float traces[1001][1001][130];
 float scratch[129], temp;
 int numchn, k1, k2, k, i, j, l, m;
 int valid_value;


 valid_value = 0;
 while ( ! valid_value)
    {
     printf ("How many channels (2-128)? ");
     scanf ("%d",&numchn);
     if (numchn < 2 || numchn > 128)
        printf ("Invalid Response, please try again\n");
     else
        valid_value = 1;
    };

 for (i=1; i <= 2; i++)
    {
     printf ("%d pass\n",i);
     for (j=1; j<= 3; j++)
        {
         printf ("Processing the %d th data set.\n", j);

/*Initialize the array traces. */

            switch (j) {
            case 1:
               {
#pragma omp parallel for private(m, l, k) shared (traces) schedule (static)
                for (m=1; m <= 1000; m++)
                   {
                    for (l=1; l <= 1000; l++)
                       {
                        for (k=1; k <= numchn; k++)
                           {
                            traces[m][l][k] = 1.0;
                           };
                       };
                   };
               };
               break;
            case 2:
               {
#pragma omp parallel for private(m, l, k) shared (traces) schedule (static)
                for (m=1; m <= 1000; m++)
                   {
                    for (l=1; l <= 1000; l++)
                       {
```

```
                        for (k=1; k <= numchn; k++)
                           {
                            traces[m][l][k] = k;
                           };
                        };
                     };
                  };
                  break;
            default:
               {
#pragma omp parallel for private(m, l, k) shared (traces) schedule (static)
                for (m=1; m <= 1000; m++)
                   {
                    for (l=1; l <= 1000; l++)
                       {
                        for (k=1; k <= numchn; k++)
                           {
                            traces[m][l][k] = numchn + 1 - k;
                           };
                        };
                     };
               };
            };

/*Calculate the medians */

#pragma omp parallel for private(m, l, k, k1, k2, scratch, temp)
shared(traces) schedule(static)
            for (m=1; m <= 1000 ; m++)
               {
                for (l=1 ; l <= 1001-numchn; l++)
                   {

/*Collect the values to be stacked */

                    for (k=1; k <= numchn ; k++)
                       {
                        scratch[k] = traces [m][l-1+k][k];
                       };

/* Perform a bubblesort */

                    for (k1=1; k1 <= numchn; k1++)
                       {
                        for (k2=1; k2 <= numchn-1; k2++)
                           {
                            if (scratch[k2] > scratch[k2+1])
                               {
                                temp = scratch[k2];
                                scratch[k2]=scratch[k2+1];
                                scratch[k2+1]=temp;
                               };
                           };
                       };
```

```
/* Find the median value and store it back into the 129th channel in traces
*/

                  k1 = (numchn +1)/2;
                  k2 = numchn - ((numchn - 1)/2);
                  traces [m][l][129] = 0.5 * (scratch[k1] + scratch[k2]);

               };
            };

         printf ("traces[1][1][129] %f\n", traces[1][1][129]);
         printf ("traces[1][800][129] %f\n", traces[1][800][129]);
         printf ("traces[1000][1][129] %f\n", traces[1000][1][129]);

        };
    };
 printf ("Hello World\n");
};
```

INTENTIONALLY LEFT BLANK.

Appendix B. Mstack FORTRAN Reference Implementation

```
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C                                                                              C
C  A simulated program for performing a Median Stack on Drag Seismic Data.     C
C                                                                              C
C  This program will assume that there are a large number of data traces,      C
C  with a constant number (M) channels per data trace.  While M can be any     C
C  positive integer, in general it is expected that 10 <= M <= 128, and        C
C  that the number of data traces and the number of measurements per trace     C
C  are both >> M.  For simplicity, the first M-1, and last M-1                 C
C  measurements per trace will be thrown away since less than M values        C
C  exist to be stacked at those points.                                        C
C                                                                              C
C  The median stack inherently involves performing a large number of sorts     C
C  on an almost infinite number of relatively small data sets (the set         C
C  size here will be N).  Therefore it does not make sense to use a            C
C  sophisticated sorting algorithm.  Instead, we will keep it simple by        C
C  using a bubblesort.  However, we will use OpenMP to allow multiple          C
C  sorts to be performed at once.                                              C
C                                                                              C
C  Ordinarily the data would be read in from a file containing partially       C
C  processed data from the field.  For our purposes, the program will          C
C  populate a large two dimensional matrix (32 bit values since                C
C  measurements have limited precision, and drag data is especially            C
C  noisy) with dummy values).  In order to minimize extraneous effects,        C
C  three different dummy data sets will be created, and the program will       C
C  cycle through the three sets twice.  The sets are:                          C
C                                                                              C
C       1)  All values will be 1.0                                             C
C                                                                              C
C       2)  The values within a single channel will be the channel             C
C           number (1 to M, where 2 <= M <= 128).                              C
C                                                                              C
C       3)  The values within a single channel will be                         C
C           float(M - channel number).                                         C
C                                                                              C
C  For simplicity, we will always use 1000 for the number of measurements     C
C  per trace, and 1000 data traces.  These values were chosen so that they     C
C  will comfortably fit in 1 GB of memory, with room left over.                C
C                                                                              C
C  NOTE:  This program does not use dynamic memory allocation.  Therefore      C
C  it is possible that if the number of channels is significantly less         C
C  than 128 that one could increase the number of data traces.                 C
C                                                                              C
C  A small number of values will be output for each data set in an effort      C
C  to prevent an optimizing compiler from optimizing away all of the work.     C
C                                                                              C
C  Written by Daniel M. Pressel at ARL in July of 2007.                        C
C                                                                              C
C  Purpose is to provide a simple reference benchmark for testing out          C
C  the ability of attached processors (e.g., FPGAs, GPGPUs, the SPEs in        C
C  the Cell processor, or Clear Speed) to accelerate floating point            C
C  applications.  This application was chosen for its simplicity.  While       C
C  it may have little relevance to the military, it comes from the             C
C  problem domain of seismic prospecting for oil, which has always been an     C
C  HPC application.                                                            C
C                                                                              C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

```
      REAL*4 traces(129,1000,1000)
      SAVE traces
      REAL*4 scratch(128)

   10 CONTINUE
      PRINT *,'How many channels (2-128)?'
      READ (5,*) numchn

      IF (numchn .LT. 2 .OR. numchn .GT. 128) THEN
         PRINT *,'Invalid Response, please try again'
         GOTO 10
      ENDIF

      DO 1010 I=1,2
         PRINT *,I,' Pass'
         DO 1000 J=1,3
            PRINT *,'Processing the ',J,'th data set'

C Initialize the array traces.

            IF (J .EQ. 1) THEN
!$OMP PARALLEL DO PRIVATE(M, L, K) SHARED (traces)
!$OMP+ SCHEDULE (static)
               DO 120 M=1,1000
                  DO 110 L=1,1000
                     DO 100 K=1,numchn
                        traces(K,L,M)=1.0
  100                CONTINUE
  110             CONTINUE
  120          CONTINUE
!$OMP END PARALLEL DO
            ELSEIF (J .EQ. 2) THEN
!$OMP PARALLEL DO PRIVATE(M, L, K) SHARED (traces)
!$OMP+ SCHEDULE (static)
               DO 220 M=1,1000
                  DO 210 L=1,1000
                     DO 200 K=1,numchn
                        traces(K,L,M)=K
  200                CONTINUE
  210             CONTINUE
  220          CONTINUE
!$OMP END PARALLEL DO
            ELSE
!$OMP PARALLEL DO PRIVATE(M, L, K) SHARED (traces)
!$OMP+ SCHEDULE (static)
               DO 320 M=1,1000
                  DO 310 L=1,1000
                     DO 300 K=1,numchn
                        traces(K,L,M)=numchn + 1 - K
  300                CONTINUE
  310             CONTINUE
  320          CONTINUE
!$OMP END PARALLEL DO
            ENDIF

C Calculate the medians
```

```
!$OMP PARALLEL DO PRIVATE(M, L, K, K1, K2, scratch, temp)
!$OMP+ SHARED (traces)
!$OMP+ SCHEDULE (static)
            DO 920 M=1,1000
               DO 910 L=1,1001-numchn

C Collect the values to be stacked

                  DO 400 K=1,numchn
                     scratch(K)=traces(K, L-1+K, M)
  400             CONTINUE

C  Perform a bubblesort

                  DO 510 K1=1,numchn
                     DO 500 K2=1,numchn-1
                        IF (scratch(K2) .GT. scratch(K2+1)) THEN
                           temp = scratch(K2)
                           scratch(K2)=scratch(K2+1)
                           scratch(K2+1)=temp
                        ENDIF
  500                CONTINUE
  510             CONTINUE


C  Find the median value and store it back into the 129th channel in traces

                  K1=(numchn + 1)/2
                  K2=numchn - ((numchn - 1)/2)
                  traces(129,L,M)=0.5 * (scratch(K1) + scratch(K2))

  910          CONTINUE
  920       CONTINUE
!$OMP END PARALLEL DO

            PRINT *,'traces(129,1,1)=',traces(129,1,1)
            PRINT *,'traces(129,800,1)=',traces(129,800,1)
            PRINT *,'traces(129,1,1000)=',traces(129,1,1000)

 1000    CONTINUE
 1010 CONTINUE
      PRINT *,'Hello World'
      END
```

Appendix C. Other Target Co-processor Classes

C.1 GPGPU

GPUs are co-processors designed specifically for processing four-dimensional floating point instructions in parallel. The four dimensions are the Red-Green-Blue-Alpha channels used in graphic texture data (see figure C-1).

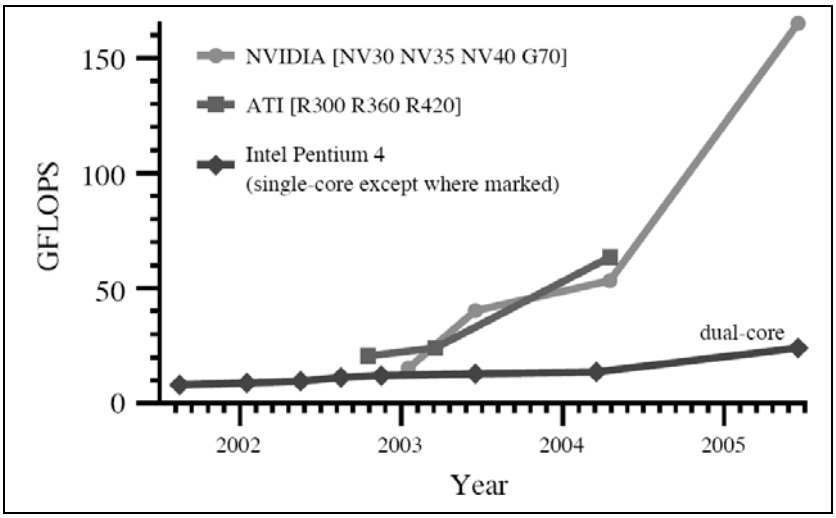

Figure C-1. Comparison of GPU and CPU floating point performance.[1]

GPUs have a pipeline consisting of one or more vertex processors, a rasterizer, one or more fragment processors, and a frame buffer. This basic architecture had remained constant over the last 20 years until recently. Now, NVIDIA and ATI have started to produce GPUs with a single unified pool of processors instead of separate vertex and fragment processors. Data related to pixel properties are stored in dedicated high speed graphics memory. This memory uses multi-banking, data streaming, specialized cache designs, and other techniques to provide extremely high bandwidth at low latency.[2]

The superior performance of GPUs over CPUs is a result of the "highly data-parallel nature of graphics computations" which "enables GPUs to use additional transistors more directly for computation, achieving higher arithmetic intensity with the same transistor count."[1]

[1]Owens, J. D.; Luebke, D.; Govindaraju, N.; Harris, M.; Krüger, J.; Lefohn, A. E.; Purcell, T. J. A Survey of General-Purpose Computation on Graphics Hardware. In *Proceedings of Eurographics*, 2005.

[2]Boggan, S. K.; Pressel, D. M. GPUs: *An Emerging Platform for General-Purpose Computation*; ARL-SR-154; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, August 2007.

GPU performance and economics are dictated by the video gaming industry which exerts a constant, tremendous pressure for better graphics processing performance. The size of the industry means that research and development costs are amortized across a huge number of retail units. In particular, this includes embedded graphics processors such as those found on gaming consoles. Every Playstation, X-box, and Wii sold includes a graphics card. In short, the existence of the video gaming industry means that GPUs have a price/performance ratio comparable (if not superior) to CPUs.

However, it is important to bear in mind the limitations of GPUs. GPUs are highly specialized devices, optimized heavily for graphics processing. Those applications whose characteristics are similar to graphics processing tasks (single precision floating point, highly data parallel, almost no sequential bottlenecks, etc.) will experience good performance acceleration from being ported to GPUs.[2]

We should soon have a GPU test bed system up on which to develop CUDA. To program this platform, we will use the CUDA API. CUDA is NVIDIA's API for programming their GPUs, primarily for non-graphical applications. It is C based but requires a massively threaded program in order to obtain high levels of performance. Massively threaded means at least 19,200 threads for the current generation of GPUs, with 256,000 threads recommended if the program is to continue to work well with future generations of their GPUs. This makes it very difficult to use for most applications, although some embarrassingly parallel applications may be able to take advantage of one or a small number of GPUs. As a result, the ability to use GPUs to reach petaflops worth of delivered performance for many applications is highly questionable.

C.2 Cellular Architectures

Another trend in HPC architecture has been the push toward "cellular architectures" (also known as "many-core" architectures), that is, arrays of simple, repeating tightly coupled processors or processor cores. Individually, each processor/core is relatively slow and consumes little power, but the architecture is designed to accommodate many of them.

The Cell Broadband Engine (BE), designed by IBM for use in Sony's Playstation gaming console, is an example of cellular architecture on a small scale. The cell has an IBM PowerPC as its main processor (for sequential sections of code), with eight "synergistic processing elements" (SPEs) for parallel computation.[*] IBM developers have ported a number of applications to Cell with good performance. For example, they achieved 25.12 Gflops per SPE doing matrix-matrix multiplication, with nearly linear parallelization. Other applications included variety of cryptography algorithms, MPEG2 (Moving Picture Experts Group) decoding, and double

[*]Note that only seven of the SPEs are consumer enabled. The eighth SPE is included as a spare to boost manufacturing yields. If a Cell chip is manufactured with a defect in one SPE, the defective SPE can be disabled and the spare SPE enabled. Furthermore, another SPE is used for operating system tasks, leaving six SPEs for actual parallel computation (Linklater, M. Optimizing Cell Core, Game Developer Magazine, April 2007).

precision Linpack.[3] However, Cell has been criticized as being very difficult to program efficiently.[4, 5] Figure C-2 gives the Cell BE's physical layout.

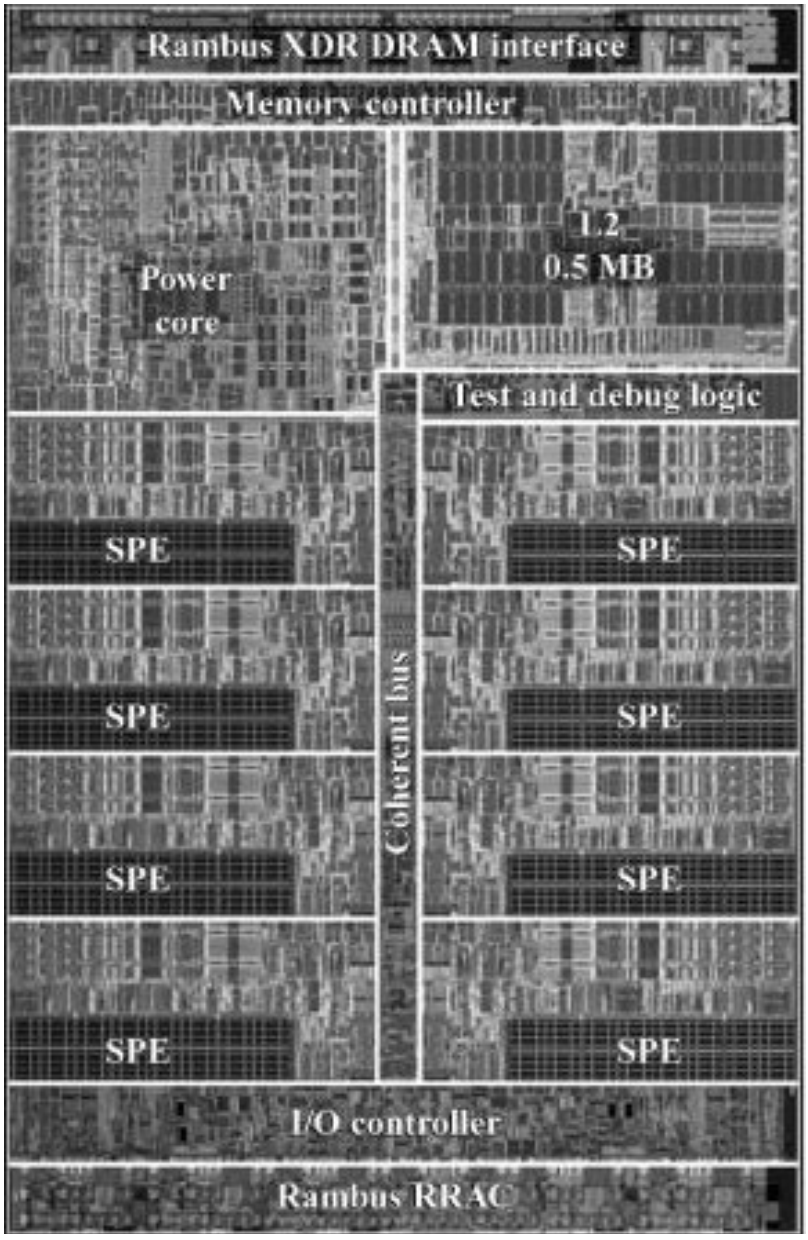

Figure C-2. The Cell BE consists of one power PC core and eight SPEs.[6]

[3]Chen, T.; Raghavan, R.; Dale, J.; Iwata, E. Cell Broadband Engine Architecture and Its First Implementation. IBM DevelopWorks, November 2005.

[4]Shankland, S. Octopiler Seeks to Arm Cell Programmers. CNet News, 22 February 2006.

[5]Scarpazza, D. P.; Villa, O.; Petrini, F. Programming the Cell Processor. *Dr. Dobb's Journal* **2007**.

[6]Kahle, J. A.; Day, M. N.; Hofstee, H. P.; Johns, C. R.; Maeurer, T. R.; Shippy, D. Introduction to the Cell Multiprocessor. *IBM Journal of Research and Development* **2005**, *49* (4/5).

Cyclops64 is an example of a cellular architecture on a much larger scale (see figure C-3). It is a "system-on-a-chip" architecture being developed by IBM. Each Cyclops64 node (chip) includes 80 processors. Each processor has two cores, each of which has its own scratch-pad memory. The two cores in each processor share a floating point execution unit. Cyclops64 chips can be combined into a larger system of as many as 13,824 nodes and 13.8 terabytes of RAM.[7] Thus, a full Cyclops64 system can support as many as 2,211,840 hardware threads. Cyclops64 uses the TinyThreads thread virtual machine and threading library.[8] Processors within a Cyclops64 node communicate via a 96- × 96-way non-internally blocking cross-bar switch. The cross-bar switch is the critical component in the design, allowing enormous bandwidth between the processors and the on-chip Static Random Access Memory (SRAM). Accesses across the cross-bar switch are uniform, except for scratch pad memory accesses (which go through the "back door," a special bus that bypasses the cross-bar switch).

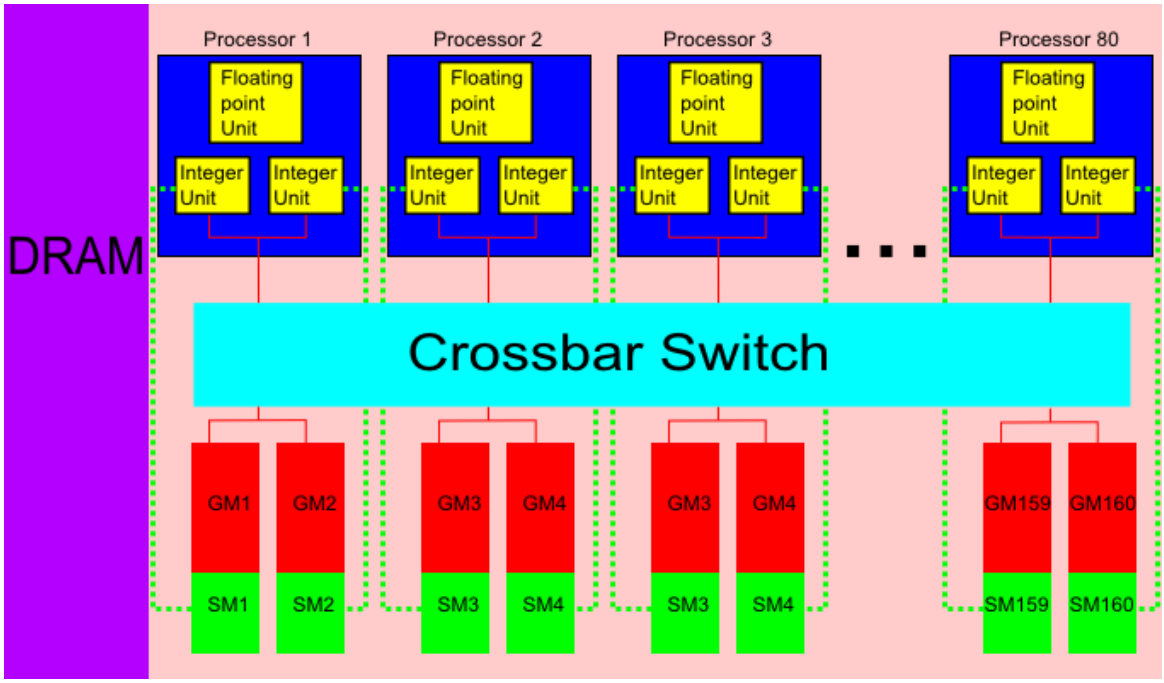

Figure C-3. A logical view of the Cyclops64 chip. (The cross-bar switch separates the dual-core processors from the SRAM memory. The memory is partitioned into global interleaved memory (GM) and processor-specific scratch-pad memory (SM). The "back door" bus is shown with a dashed line.)

[7] Zhang, Y. M. P.; Jeong, T.; Chen, F.; Wu, H.; Nitzsche, R.; Gao, G. R. A Study of the On-Chip Interconnection Network for the IBM Cyclops-64 Multi-Core Architecture. In the *Proceedings of 20th International Parallel and Distributed Processing Symposium*, 2006.

[8] Del Cuvillo, J.; Zhu, W.; Hu, Z.; Gao, G. R. FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture. Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the *32nd Annual International Symposium on Computer Architecture*, 2005.

Cyclops64 is currently in the final stages of development and should be released in late 2007 or 2008. Because the architecture does not yet physically exist, our evaluation of its performance will be based on simulation using the Functionally Accurate Simulation Toolset (FAST) simulator.[9] FAST "is an execution-driven, binary-compatible simulator of a multichip multithreaded C64 system. It accurately reproduces the functional behavior and count of hardware components such as thread units, on-chip and off-chip memory banks, and the 3D-mesh network."

C.3 On FPGA Using DIME-C

Another potential platform for Mstack is on FPGAs with the DIME-C "C-to" language. Nallatech offers the DIME-C and DIMETalk* software development tools to ease the process of programming applications on an FPGA. DIME-C is the high-level language used by Nallatech's software for VHDL translation. An attractive feature of DIME-C is the fact that it is a subset of American National Standards Institute (ANSI) C with the identical syntax. A user only needs to learn the portion of standard C language that is not supported in DIME-C. This allows a minimal learning curve to start coding in DIME-C. Since the language is a subset of ANSI C, DIME-C code can also be debugged with the use of any ANSI C compilers. Hardware optimizations exploited through parallelism and pipelines are automatically applied by the compiler without user intervention. When the DIME-C code is compiled, a graphical representation showing parallel and pipelined structure is generated.

Currently, DIME-C and DIMETalk are supported only on a Windows† platform. Since a large amount of system memory is used during the synthesis process executing Xilinx software, 64-bit Windows or Linux operating systems are recommended for software installation. The FPGA's internal and external interfaces are handled by the DIMETalk software. DIMETalk divides and represents an overall FPGA design as separate individual components with pre-defined interconnections. For example, a clock driver, a host interface, a memory element, and user design modules are the minimum components required for an FPGA design. The following list describes the minimum components of a DIMETalk network:

- Clock driver provides the clock signals for a design.
- Host interface defines the FPGA board interface to a host system allowing communication to an FPGA.
- Memory elements can store data and are visible and accessible by a host.
- User module defines an algorithm in hardware.

---

[9] Del Cuvillo, J.; Zhu, W.; Hu, Z.; Gao, G. R. TiNy Threads: A Thread Virtual Machine for the Cyclops-64 Cellular Architecture. Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the *19th International Parallel and Distributed Processing System*, 2005.

* DIMETalk is a trademark of Nallatech.

† Windows is a trademark of Microsoft Corporation.

Having the host interface component greatly simplifies the implementation of FPGA and host communication by hiding the underlying details associated with PCI bus and FPGA pin connections. Basically, DIMETalk provides a work space for placing and connecting elements within a design, creating a network inside an FPGA.

The FUSE API functions are used for a host to communicate with an FPGA board. A standard C code, referred as a host C file, is written that includes the API functions. The host C file is executed on the host side, which will load, control, and execute the hardware design on an FPGA. A generic host C file is created by DIMETalk, which relieves a developer from the details of initial FPGA setup. However, a user is expected to include I/O data transfers between the host system and an FPGA to the generic host C file. Because the read and write FUSE API functions can only operate on memory elements, a processing unit within an FPGA is responsible for placing outcome results to these memory elements in order for the host to read the final results.

The tools offered by Nallatech attempt to simplify the process of hardware design by offering a high-level language DIME-C, providing a PCI-X interface component, and generating a host C file. Furthermore, hardware optimizations are automated by the DIME-C compiler. Unfortunately, these user-friendly features result in a limited amount of hardware design flexibility for exploiting parallelism. For example, the conversion from DIME-C to VHDL creates ambiguity problems, such as with addition, which can be instantiated in hardware with a carry-propagate adder, a carry-save adder, or a carry-look-ahead adder. Furthermore, ability to control low level signals and architecture to achieve hardware acceleration is lost. Since the DIME-C compiler is in charge of designing hardware for an algorithm written in DIME-C, optimizations are limited to the operations performed by the DIME-C compiler.

The DIME-C compiler derives a hardware design from a subset of standard C, which is a procedural language. A procedural language is intended for programming processors and is fundamentally different from designing hardware. Although the intentions and benefits of using DIME-C for programming FPGAs are attractive, achieving performance improvement directly from a language targeted for a CPU can be a limiting factor.

NO. OF
COPIES ORGANIZATION

1 (PDF ONLY) DEFENSE TECHNICAL INFORMATION CTR
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FORT BELVOIR VA 22060-6218

1 US ARMY RSRCH DEV & ENGRG CMD
SYSTEMS OF SYSTEMS INTEGRATION
AMSRD SS T
6000 6TH ST STE 100
FORT BELVOIR VA 22060-5608

1 DIRECTOR
US ARMY RESEARCH LAB
IMNE ALC IMS
2800 POWDER MILL RD
ADELPHI MD 20783-1197

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRD ARL CI OK TL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

1 DIR USARL
AMSRD ARL CI OK TP (BLDG 4600)

NO. OF COPIES ORGANIZATION

1 PROGRAM DIRECTOR
C HENRY
1010 N GLEBE RD
STE 510
ARLINGTON VA 22201

1 DEPUTY PROGRAM DIRECTOR
L DAVIS
1010 N GLEBE RD
STE 510
ARLINGTON VA 22201

1 HPC CENTERS PROJ MGR
B COMMES
1010 N GLEBE RD
STE 510
ARLINGTON VA 22201

1 CHIEF SCIENTIST
D POST
1010 N GLEBE RD
STE 510
ARLINGTON VA 22201

1 ADELPHI LAB CTR
AMSRD ARL CI
J GOWENS II
BLDG 205 RM 3A012C
ADELPHI MD 20783-1197

1 DIRECTOR
AMSRC ARL CI HC
M LEE
BLDG 1425 RM 805
FORT DETRICK
FREDERICK MD 21702-5000

1 J OSBURN
CODE 5594
BLDG A49 RM 15
4555 OVERLOOK RD
WASHINGTON DC 20375-5340

1 DIRECTORATE
AIR FORCE RSRCH LAB
MATRLS & MFG
R PACHTER
AFRL/MLPJ
BLDG 651 RM 189
3005 HOBSON WAY
WRIGHT-PATTERSON AFB OH
45433-7702

NO. OF COPIES ORGANIZATION

1 AIR FORCE RSRCH LAB
K HILL
AFRL/SNS BLDG 254
2591 K ST
WRIGHT-PATTERSON AFB OH
45433-7602

1 R LINDERMAN
AFRL/IF
525 BROOKS RD
ROME NY 13441-4505

1 US ARMY AEROFLIGHTDYNAMICS
DIRECTORATE
US ARMY AFDD (RDEC)
R MEAKIN
AMES RESEARCH CTR M/S T27B-1
MOFFETT FIELD CA 94035-1000

1 ARMY RSRCH OFC
AMSRD ARL RO EN
A RAJENDRAN
PO BOX 12211
RESEARCH TRIANGLE PARK NC
27709-2211

1 NVL OCEANOGRAPHIC OFC
OFC OF THE TECH DIR
J HARDING
CODE OTT
STENNIS SPACE CENTER MS 39529

1 INFO TECHLGY LABS
ARMY ENGINEER RSRCH & DEV CTR
D RICHARDS
VICKSBURG MS 39810

1 SPAWAR SYS CTR
C PETERS
53360 HULL ST
BLDG 606 RM 318
SAN DIEGO CA 92152

1 ARNOLD ENGRG DEV CTR
C VINING
1099 SCHRIEVER AVE
STE E205
ARNOLD AFB TN 37389

1 AIR FORCE RSRCH LAB
SENSORS DIRCTRT
T WILSON
2241 AVIONICS CIR
WRIGHT-PATTERSON AFB OH 45433

NO. OF
COPIES ORGANIZATION

1 US ARMY RSRCH & DEV CTR
NVL CMND CTRL & OCEAN
SURVEILLANCE CTR
HPC COORDNTR & DIR
DOD DISTRIBUTED CTR
NCCOSC RDTE DIV D3603
L PARNELL
49590 LASSING RD
SAN DIEGO CA 92152-6148

1 UNIV OF TENNESSEE
ASSOC DIR
INNOVATIVE COMPUTING LAB
CMPTR SCI DEPT
S MOORE
1122 VOLUNTEER BLVD STE 203
KNOXVILLE TN 37996-3450

7 COMPUTER ARCHITECTURE &
PARALLEL SYSTEM LAB
M PELLEGRINI
G GAO
I VENETIS
J MANZANO
P ZHANG
D OROZCO
G TAN
322 DUPONT HALL
NEWARK DE 19716

ABERDEEN PROVING GROUND

17 DIR USARL
AMSRD ARL CI H
C NIETUBICZ
AMSRD ARL CI
R NAMBURU
AMSRD ARL CI HC
P CHUNG
J CLARKE
S PARK
D PRESSEL
D SHIRES
R VALISETTY
C ZOLTANI
AMSRD ARL CI HM
P MATTHEWS
R PRABHAKARAN
AMSRD ARL CI HS
D BROWN
T KENDALL
K SMITH

NO. OF
COPIES ORGANIZATION

AMSRD ARL WM BC
K HEAVEY
J SAHU
P WEINACHT

INTENTIONALLY LEFT BLANK.