Speedup using Flowpaths for a Finite Difference Solution of a 3D Parabolic PDE

Darrin M. Hanna[1], Anna M. Spagnuolo[2], and Michael DuChene[3]

[1]Oakland University
Dept. of Comp. Sci. & Engineering
Rochester, MI 48309
dmhanna@oakland.edu

[2]Oakland University
Dept. of Mathematics & Statistics
Rochester, MI 48309
spagnuol@oakland.edu

[3]Oakland University
Dept. of Comp. Sci. & Engineering
Rochester, MI 48309
mjduchen@oakland.edu

Abstract

Partial differential equations (PDEs) are used to model physical phenomena and then appropriate convergent numerical algorithms are employed to solve them and create computer simulations. In many important applications, such as weather prediction and contaminant transport processes, simulation outputs are required in real time or even faster, yet the spatial component of the problem is very large, thereby increasing the computational time. In addition, often times numerical scientists work in groups to create a large-scale code, but they work individually on PCs to test components of the code, so that speedup of the computational algorithms on PCs is desirable. There is a benefit to creating and using custom hardware to perform the numerical calculations faster than commodity hardware. This work uses a high-level programming language (Java) to behaviorally describe, and then implement, a finite difference solution of a parabolic PDE as a custom hardware circuit targeted to an FPGA. The results show that the circuits can perform the calculations 1 to 2 orders of magnitude faster than commodity hardware.

1. Introduction

Numerical methods are extensively used in the sciences and engineering for simulating models of physical phenomenon. These numerical simulations can be very complex, often taking days or even months to complete or requiring expensive computing clusters. There is always a benefit of speeding up the calculations to solve more complex problems in less time. Another benefit of speeding up calculations is to solve problems in real time that could not otherwise keep up.

Using direct hardware methods to achieve speedup for calculations can present two major benefits. A low cost, portable solution would result if the slowest part of the simulation was implemented in custom hardware while the remainder of the algorithm was left to execute on a regular processor. The low cost would make this accessible to a wide audience, while the portability would be useful for mobile applications. If cost or portability is not a concern then more hardware, in terms of a high-density large array of chips, could be allocated, resulting in even more speedup.

Special purpose processors (SPPs) can be used to achieve speedup in computing. Some examples of using an SPP for performing specific functions quickly include a math co-processor in a CPU for performing floating point operations, a graphics card in a personal computer (PC), and a hardware encryption co-processor in an internet router. Since SPPs are made to perform a specific function, they do not suffer from the overhead that a general-purpose processor (GPP) does. SPPs can execute an algorithm much more quickly than a GPP and at much lower frequencies therefore requiring far less power. Despite these great benefits, a major roadblock to creating SPPs is that designing them is a tedious task that can take months to accomplish and require extensive expertise in hardware design. Thus, for every part of a numerical algorithm an SPP would have to be designed by an expert to achieve speedup. This would be very expensive and could take weeks, months or years.

Furthermore, the SPP design for a specific part of a numerical algorithm would not necessarily work for anything else; a new SPP would have to be designed as new codes are made. Clearly, this major roadblock makes it impractical to take advantage of great speedup using hardware for numerical methods. Therefore, a

This work supported by NSF grant #0636895.

method to automatically generate SPPs from programs written in a high-level language that is easy to use, requires more basic skills, and generates SPPs that can be executed at usefully high frequencies would overcome this roadblock. In this way, numerical code could automatically generate SPP hardware in minutes and execute much more quickly.

The High-Performance FPGA Laboratory (HPFL) at Oakland University, under the direction of Dr. Darrin Hanna, has developed such a hardware architecture called *Flowpaths*. Flowpaths are custom made circuits that directly implement an algorithm in hardware; they are a class of SPPs. The HPFL has developed a compiler that is able to convert programs written in the Java high-level programming language to flowpaths, for direct implementation as hardware. Thus, overcoming lengthy design times for creating high-performance SPPs, they can be generated in minutes rather than months. More importantly, this is done with little to no sacrifice in performance. Alternative methods for generating hardware from a high-level language have previously been researched and developed but still suffer from performance-limiting problems caused by inherent routing bottlenecks and unnecessary execution cycle overhead. Because of unique characteristics, flowpaths overcome these problems and have actually been shown to perform within a factor of 2 of equivalent hand-crafted circuits.

Flowpaths are described in the standard hardware description language VHDL. This VHDL code is used to implement hardware solutions on reprogrammable logic devices such as FPGAs and the code can also be used to fabricate ASICS. The VHDL is generated automatically from Java programs using our compiler. The compiler takes as input Java bytecodes generated from a Sun Microsystems compliant Java compiler. The ability to exploit and optimize flowpath techniques to compile numerical codes would profoundly impact researchers' ability to run numerical simulations in less than one-tenth of the time required when compared with running it on a PC.

In this work, we have taken a finite difference code for the solution of a three-dimensional parabolic PDE in time on a rectangular box domain, compiled it to flowpaths, and simulated it. Using flowpaths, we achieved a speedup of 700 times over the Java version, 480 times over the FORTRAN90 code and a 60 times speedup over the C++ version run on a PC. Section 2 gives a brief introduction to flowpaths. Section 3 describes the numerical method. A description of our experimental results can be found in Section 4. Finally, conclusions and future work are given in Section 5.

2. Flowpaths

Generating a flowpath from Java bytecode can be demonstrated with a simple example. Consider the Java code for one of Euclid's GCD algorithms shown in Listing 1 and the corresponding Java bytecodes in Listing 2.

Listing 1: Euclid's GCD algorithm in Java

```
1:  int gcd (int x, int y)
2:  {
3:    //Euclid's GCD
4:    while (x != y)
5:    {
6:      if (x<y)
7:         y -= x;
8:      else
9:         x -= y;
10:   }
11:   return x;  }
```

Listing 2: Java Bytecodes for Euclid's GCD

```
0:  goto 19
3:  iload_1       //Push x
4:  iload_2       //Push y
5:  if_icmpge 15  //if x>=y goto 15

8:  iload_2       //Push y
9:  iload_1       //Push x
10: isub     //y-x
11: istore_2  //Store new y
12: goto 19

15: iload_1       //Push x
16: iload_2       //Push y
17: isub     //x-y
18: istore_1  //Store new x

19: iload_1  //Push x
20: iload_2  //Push y
21: if_icmpne 3 //if x!=y goto 3

24: iload_1  //Push x
25: ireturn
```

In general, algorithms can be considered as a set of operations, data, and a controller to determine execution order. Others have described methods to convert such programs into special-purpose processors where each variable is implemented as a register [1, 2]. Using these other methods, consider the hardware that would be generated from these bytecodes. It has several drawbacks including unnecessary states. This hardware is shown in Figure 1. Each block is a digital component that performs the function as labeled. The inputs and outputs are data busses. Each of the components is executed according to the state diagram in Figure 2.

Next to each component and state, the line that matches Listing 2 is annotated accordingly. A simple optimization can be made by substituting the two *isub* components with a single *isub* component and a multiplexer as shown in Figure 1.

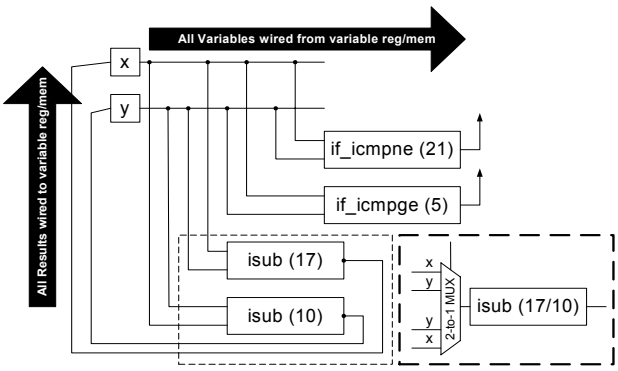

Figure 1: Hardware implementing the GCD bytecode

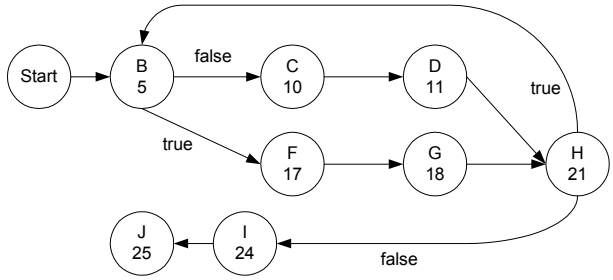

Figure 2: A state diagram to control the execution path of the hardware in Figure 1

Although this is a possible approach to generating hardware directly from software applied to an intermediate representation that is primarily stack-based, it is important to note a general problem which will become very large, quickly. Since the Java bytecode representation uses local variables that are implemented as registers, the inputs to the operations must be routed from the registers or memory storing the variables and the results must be stored back into the register or memory. This presents a significant routing problem. Furthermore, critical path times increase and the maximum execution frequency decreases significantly.

Using a stack-based intermediate representation where local variables are stored on a stack and operations are done to the top elements of the stack, with results left on top of the stack, data flow can be streamlined and the problem eliminated. By placing local variables on a stack, stack manipulation words will frequently be used to move elements around before and after executing operations, but result-loading instructions can be eliminated. This presents a significant advantage while using flowpaths since stack manipulation words translate into zero-clock-cycle routing.

In Java, local variables are kept in a local variable array resulting in numbered variables. These variables are loaded into the operand stack and loaded with successive results. Since all local variables are 'passed through' registers in each OP, successive *iload_*x_i instructions are translated into copies of the wires corresponding to the x_i's routed in the proper order for input into the next OP.

One can implement stack-based code, e.g. Java bytecodes, with a local variable array and achieve several orders of magnitude faster performance without the routing problem described when using the variable-to-register methods for special-purpose processors. Flowpaths significantly increase the maximum execution speed and in most cases distributes routing reasonably. Standard OPs that are pre-designed with generic bus-width and with a variable number of 'pass-through' registers form a flowpath standard library. These standard OPs perform basic instructions, such as *imul, iadd,* and *if_icmpne*. Additionally, custom OPs can be designed at a hardware or software level and stored in this library where they will be considered standard words by the compiler. This allows one to invoke methods that aren't developed in Java but are OPs that follow the standard *reset-done* flowpath paradigm.

The flowpath to compute the greatest common divisor requires three OPs: An equality detector (OPEq), a magnitude comparator (OPLt), and a subtraction OP (OPMinus). A path using multiplexers and demultiplexers connects these OPs and a simple state-machine controller controls the entire circuit. Each of these OPs requires only one clock-cycle to execute.

Figure 3 shows the GCD algorithm in Listing 1 implemented as an optimized flowpath. The formal algorithm for compiling Forth code to flowpaths, an initial unoptimized GCD flowpath, and optimization techniques are detailed in Hanna (2003) [3]. The data are 32-bit integers. Characteristic of a multi-cycle OP, the GCD OP itself has a *reset* signal and *done* signal.

When the *reset* signal is brought low, the OP begins execution and when execution has completed, the *done* signal is asserted high. Figure 4 shows the state transition diagram for the flowpath controller. Since the GCD function requires more than one clock cycle to execute, the *rst* signal must be brought low to begin execution. Execution begins in the *Start* state. After the final state, *F*, the state machine returns to the *Start* state and will begin again unless the *reset* signal is returned to high.

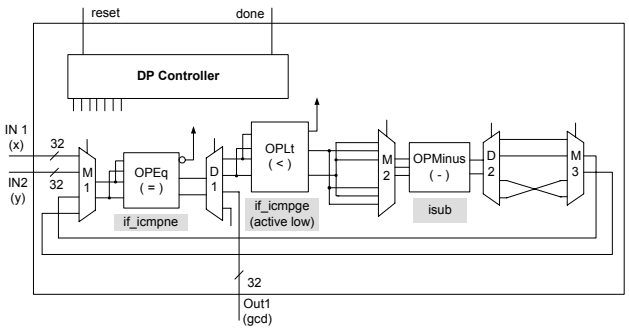

Figure 3: Flowpath datapath for GCD

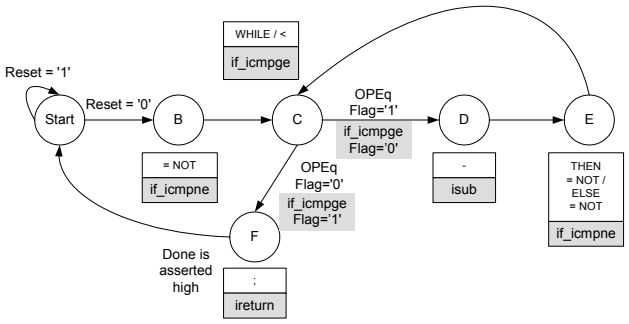

Figure 4: Flowpath controller for GCD

Flowpaths in general support delays, loops, branches, memory access, basic recursion, and other programmatic constructs. Using flowpaths requires less space and has a maximum clock frequency much higher than the FPGA implementation of the GCD implemented using Handel-C [4]. Flowpaths can outperform microprocessors at lower clock frequencies and therefore consume less power than microprocessors or microprocessor cores. Flowpaths can be partitioned into subsets that can take advantage of runtime dynamic partial reconfiguration [5].

3. Numerical Solution of a 3D Reaction-Diffusion Equation using a Parabolic PDE

The numerical code that we are implementing using flowpaths is a finite difference code for solving the diffusion equation with a source term in a domain Ω. Let c = c(x, y, z, t) be the concentration of the unknown, let D be the diffusion of the medium, and let ρ be the growth constant of the material. The governing equation for the flow is given by

$$\frac{\partial c}{\partial t} + \nabla\bullet(uc) - \nabla\bullet(D\nabla c) = \frac{f(c)}{\phi}$$

where $f(c)/\phi = \rho c + S(c)$, is the growth rate and the source/sink, ϕ is the porosity, and u is the velocity (which we take to be zero for this study).

The Dirichlet and Neumann boundary conditions are imposed as follows:

$$u\bullet v = D\nabla c\bullet v = 0, \text{ on } \partial\Omega,$$

And the initial condition is prescribed as

$$c(x, 0) = c_{init}(x), \text{ in } \Omega.$$

The numerical solution of the PDE begins with a discretization in each of the three spatial variables and in time. Here, we use a cell-centered finite difference method with discretization parameter h in x, y, and z for the unknown computational solution c^*.
Let i = (ih, jh, kh), and define

$$f_{\mathbf{i}}^* = f((i+1/2)h,\ (j+1/2)h,\ (k+1/2)h),$$
$$c_{\mathbf{i}}^* = c((i+1/2)h,\ (j+1/2)h,\ (k+1/2)h),$$

and the diffusion coefficient across the faces of the rectangles in the grid as follows:

$$D_{\mathbf{1,i}} = D(ih, (j+1/2)h, (k+1/2)h),$$
$$D_{\mathbf{2,i}} = D((i+1/2)h, jh, (k+1/2)h),$$
$$D_{\mathbf{3,i}} = D((i+1/2)h, (j+1/2)h, kh).$$

We use the following discretization for the diffusion term:

$$(\nabla\bullet D\nabla c^*)_{h,\mathbf{i}} = \frac{1}{h^2}\sum_{l=1}^{3}(D_{l,\mathrm{i}+h\mathbf{e}_l}(c^*_{\mathrm{i}+h\mathbf{e}_l} - c^*_{\mathrm{i}}) - D_{l,\mathrm{i}}(c^*_{\mathrm{i}} - c^*_{\mathrm{i}-h\mathbf{e}_l})$$

Then, the finite difference equation can be written as

$$\frac{c_{\mathbf{i}}^{*,n} - c_{\mathbf{i}}^{*,n-1}}{\Delta t} - (\nabla\bullet D\nabla c^{*,n})_{h,\mathbf{i}} = \frac{f_{\mathbf{i}}^{*,n}}{\phi} \text{ on } \Omega_h,$$

where Ω_h is the discretized domain with the analogous boundary and initial conditions. We use an operator splitting method to solve the problem by separating it into transport and diffusion as follows:

Transport: We assume the special case that $u = 0$.

$$\overline{c}_{\mathbf{i}}^{*,n} = e^{\rho\Delta t} c_{\mathbf{i}}^{*,n-1}$$

Diffusion:

$$\frac{c_{\mathbf{i}}^{*,n} - \overline{c}_{\mathbf{i}}^{*,n}}{\Delta t} - (\nabla\bullet D\nabla c^{*,n})_{h,\mathbf{i}} = S_{\mathbf{i}}^{*,n}$$

Solving the transport step is straightforward. We use the Conjugate Gradient Method preconditioned by the diagonal to solve the diffusion step iteratively. Then we iterate over transport and diffusion at each time step.

An application of our PDE is in tracing neurochemicals in the brain [6]. To that end, the shape of the boundary, $\partial\Omega$, was selected to be an approximation of the boundary of a rat brain created using MRI slices.

4. Experimental Results

The numerical code was written in FORTRAN90, Java, and C++ for benchmarking purposes. All codes were run on a PC. The CPU used in the benchmarking was a 1.10 GHz Intel Pentium M Dothan (90 nm, Family 6, Model D, Stepping 6, Revision B1) running in normal state with 1.25 GB of RAM. No optimizations were done to the compiled code to make use of the processor's SIMD instructions.

The Java code was compiled into Java bytecode using Sun Microsystems Standard Java Compiler, version 1.5.0_08. The bytecode was compiled into flowpath circuits described in VHDL using the flowpath compiler. Xilinx ISE 8.1.03i was used to synthesize and implement the flowpaths. The development board targeted was the Xilinx XUP Virtex II Pro Development System which includes a Xilinx Virtex2 XC2VP-30 FPGA.

The flowpath clock cycle counts were obtained using Mentor Graphics ModelSim SE 6.2b. In any cases where the running time of the flowpath is displayed, it is assumed that the flowpath is running at 100 MHz, unless otherwise specified. The actual maximum clock frequency of the flowpath was 97.675 MHz, based on the synthesis reports. The frequency was limited by the un-optimized floating point addition operation that was used. Once this operation is optimized, the minimum clock period would reach approximately 7 ns, which would result in a clock frequency of about 140 MHz. This is a sufficiently fast frequency for testing purposes, because the actual system architecture considered assumed that the flowpath would be communicating with the DDR memory over the IBM CoreConnect on-chip peripheral bus (OPB) which has a maximum frequency of 125 MHz [7]. During simulation, it was assumed that the bus introduced an average of 1 additional clock cycle of latency when accessing the memory to account for either a synchronous bus connection or asynchronous bus arbitration.

Table 1 shows the running time comparison between the different platforms. Obviously, as the grid becomes finer and the number of points increases, the running time increases by a proportional factor.

Table 2 shows a comparison of the clock cycle count among the different platforms. The clock cycle counts for the CPU were extrapolated from the algorithm run time. The Pentium is a superscalar processor and it can be difficult to get exact clock cycle counts.

Table 3 shows the speedup between each of the implementations including three different experiments with increasing number of points. In each case, the flowpath is considered the ideal (unit) case. As the number of points increase, C++ scales almost the same as the flowpath. FORTRAN is second-best in terms of scaling and Java is the worst since the Java Virtual Machine (JVM) introduces additional overhead. This table does not imply that flowpaths or C++ scale linearly with the algorithm size; it is a comparison to the flowpath. Figure 5 shows the runtime as a function of the relative algorithm size (a multiple increase in the number of points). While runs with more points require more time from Java, Fortran, and C++, the increase in time required from the flowpath is much smaller since the flowpath does not have load-execute-store overhead or worse, that of the JVM.

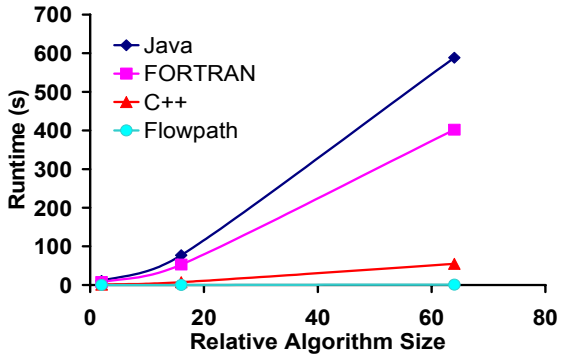

Figure 5: Runtime vs. relative algorithm

Table 1: Runtime for each implementation

# of Points	Algorithm Runtime (milliseconds)			
	CPU – Java	CPU – C++	CPU – FORTRAN	Flowpath
1650	10,405	1,018	7,311	16
13200	76,991	7,202	52,545	112
105600	588,186	54,705	401,978	835

Table 2: Clock cycle counts for each implementation

# of Points	Clock Cycle Count (#)			
	CPU - Java	CPU - C++	CPU - FORTRAN	Flowpath
1650	1,445,500,000	1,119,552,500	8,042,100,000	1,584,548
13200	4,690,100,000	7,922,030,600	57,799,500,000	11,157,689
105600	47,004,600,000	60,176,010,400	42,175,800,000	83,545,716

Table 3: Speedup relative to the Flowpath

# of Points	Flowpath Speedup			
	CPU - Java	CPU - C++	CPU - FORTRAN	Flowpath
1650	657	64	461	1
13200	690	65	471	1
105600	704	65	481	1

Depending on the target technology, the flowpath may require more space than available on the FPGA. In that case, it may be desirable to implement only part of the code as a flowpath while the remaining code executes on a microprocessor core. FPGAs such as the Xilinx Virtex 2 Pro family of chips contain embedded processor cores that can interact with the reconfigurable FPGA resources for implementing this type of cooperative scheme. Of course, since some of the code is executed on the processor, the speedup is reduced. Speedup can be maximized by identifying the code that brings the most overhead to the program and implementing those as flowpaths on the available FPGA real estate.

To that end, the code was arbitrarily partitioned into a reasonable number of functions or modules to help identify bottlenecks in the code. Table 4 shows the Java code profile and Table 5 shows the C++ code profile. According to the results shown in Table 4, the partition chosen reveals an opportunity to significantly improve performance by implementing even just one of the first two functions listed in hardware. Often, a portion of code what is responsible for much of the processing overhead does not require a proportionate amount of space to implement using flowpaths. Instead, much less space is required if the function contains little code.

Partitioning the algorithm to implement bottleneck code portions as flowpaths and other portions to run on an embedded core has another advantage. These separate functions could also be implemented as flowpaths on different FPGAs wired together. The top-level state machine could even be implemented on its own FPGA with the necessary wires connected to the other FPGAs for the control logic. Indeed, flowpaths resolve data dependency issues which makes them naturally suited to this type of architecture. The number of I/O must also be considered. In fact, a similar architecture exists that uses only 2 FPGAs [5]. This is an advantage of flowpaths over using other methods, such as Handel-C, for generating hardware from software.

Table 4: Java code profile

Java	
Function	Count
doit2	54.76%
doit2a	42.66%
doit3a	1.38%
iterate	0.50%
doit4	0.42%
doit1a	0.12%
doit1	0.05%
doit1b	0.05%
brainy	0.02%

Table 5: C++ code profile

C++	
Function	Count
doit2a	40.00%
doit2	23.10%
doit3a	20.30%
iterate	8.20%
doit4	8.00%
alloc3d	0.20%
doit1a	0.10%
doit1b	0.00%
mulit2	0.00%

5. Conclusions and Future Work

This work has shown how flowpaths make it possible to execute a numerical algorithm in less running time than on conventional computers, up to two orders of magnitude faster than a computer. Flowpaths are able to achieve this performance by utilizing large amounts of chip area; however, the modularity of flowpaths naturally allows them to be split into smaller circuits for easier implementation when space is a concern.

There are many opportunities to increase this performance gain. These include optimizing floating-point operations, exploiting constructs that are common to many numerical codes to optimize flowpaths specifically for speeding up numerical codes, and implementing numerical codes using parallel flowpaths. Much remains to be explored in terms of limitations from large data sets that require significant memory, ways to best partition algorithms to maximize the value of limited FPGA space, and characterizing the relationship between the numerical code compiled to flowpaths and the FPGA space required. Currently, we are exploring these opportunities, challenges, and limitations using several different well-known numerical methods and applications.

6. References

[1] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*: John Wiley & Sons, Inc., 2002.

[2] J. M. P. Cardoso and H. C. Neto, "Compilation for FPGA-Based Reconfigurable Hardware," *IEEE Design & Test of Computers Magazine*, vol. 20, pp. 65-75, 2003.

[3] D. M. Hanna, "A Novel Method for Generating Microprocessor-less Systems with Applications in Bioengineering," in *Dept. of Comp. Science and Eng*. Rochester, MI: Oakland University, 2003.

[4] D. M. Hanna and R. E. Haskell, "Flowpaths: Compiling Stack-Based IR to Hardware," *Microprocessors and Microsystems*, vol. 30, pp. 125-136, 2006.

[5] D. M. Hanna and M. Duchene, "Executing Large Algorithms on Low-Capacity FPGAs using Algorithm Partitioning and Runtime Reconfiguration," *Microprocessors and Microsystems*, in press 2007.

[6] K. R. Swanson, C. Bridge, J. D. Murray, and E. C. A. Jr., "Virtual and real brain tumors: using mathematical modeling to quantify glioma growth and invasion," *Journal of the Neurological Sciences*, vol. 216, pp. 1-10, 2003.

[7] Xilinx Inc., "OPB Usage in Xilinx FPGAs," September, 2005.