Application-guided Tool Development for Architecturally Diverse Computation

Roger D. Chamberlain
Jeremy Buhler
Mark A. Franklin
James H. Buckley

Roger D. Chamberlain, Jeremy Buhler, Mark A. Franklin, and James H. Buckley, “Application-guided Tool Development for Architecturally Diverse Computation,” in *Proc. of Symposium on Applied Computing,* March 2010, pp. 496-501.

Dept. of Computer Science and Engineering
Washington University in St. Louis

Application-guided Tool Development for Architecturally Diverse Computation

R.D. Chamberlain, J. Buhler, M. Franklin
Dept. of Computer Science and Engineering
Washington University in St. Louis
{roger,jbuhler,jbf}@wustl.edu

J.H. Buckley
Dept. of Physics
Washington University in St. Louis
buckley@wustl.edu

ABSTRACT

Architecturally diverse computation exploits non-traditional computing platforms (e.g., field-programmable gate arrays, graphics processors, heterogeneous chip multiprocessors) to execute user applications. We have designed the Auto-Pipe tool set with the goal of easing the task of developing applications for architecturally diverse systems. Prior to and during the course of Auto-Pipe's design, we have developed a number of real, substantial applications, and the the lessons learned during the development of these applications has had a direct bearing on the capabilities of Auto-Pipe. In this paper, we describe the relationship between our application development experience and Auto-Pipe. In short, how have applications guided the tools' evolution and development?

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Monte Carlo*; J.2 [**Physical Sciences and Engineering**]: Astronomy; J.3 [**Life and Medical Sciences**]: Biology and genetics

General Terms

Algorithms, Design, Languages

Keywords

Monte Carlo simulation, encryption, computational finance, computational biology, computational astrophysics, approximate text search, application development tools

1. INTRODUCTION

For a number of decades, computing performance gains were regularly achieved by rapidly increasing clock frequencies. This trend has recently slowed, and alternative approaches to performance improvement are now required. One approach that has significant promise is the use of alternative computing engines, such as field-programmable gate arrays (FPGAs) and graphics processing units (GPUs), to execute portions of an application, typically in conjunction with general-purpose processors. This approach is especially well suited to applications that process streaming data, such as from a sensor array from a telescope or other scientific instrument. We refer to such systems as architecturally diverse computing systems, also referred to as hybrid systems.

The embedded systems design community has been an early adopter of architecturally diverse computing, especially given the additional constraints on weight, power, and volume that are typically present in embedded system designs. However, widespread utilization of diverse computing has been somewhat elusive. Each platform has its own unique characteristics, both in terms of architecture and application development process. In addition, deploying portions of applications across multiple computing components requires that the different components coordinate their efforts. This imposes additional difficulties on the application developer, as these disparate subsystems are not typically well integrated into a coherent whole.

Many of the issues described above can be addressed by altering the state of practice for application development on diverse systems. In what follows, we presume that applications are expressed, either originally by their developers or as a result of some (semi-) automatic transformation, as a collection of articulated *kernels* that execute concurrently and communicate via explicit messages. Essentially, this is the streaming computation model.

The above approach is hardly new. Lee [15] has argued that coordination languages represent a better mechanism for reasoning about concurrency than traditional thread-based approaches. Additional languages that fit this model include Brook [3], Impulse C [17], StreamC/KernelC [5], StreamIt [19], and Streams-C [10]. Common to all of the above is the use of dataflow semantics between "kernels" or "blocks," undecomposable computations that are to be mapped to individual computing components.

Our extension to the above idea is to generalize the set of computing components that can be targets for the deployment of kernels. Auto-Pipe [7] is a tool set that supports application development of streaming computations on architecturally diverse systems. Auto-Pipe enables an application developer to: 1) author the application using the language(s) of his/her choice for expressing kernel computations and a coordination language, called X, to express the streaming topology; 2) model the application using a

SAC'10 March 22-26, 2010, Sierre, Switzerland.

federated simulation system [8]; and 3) deploy the application on a diverse architecture [4].

During initial development and over the course of its existence the design of the Auto-Pipe tool set has been influenced significantly by a group of real applications that span embedded systems, computational science, and unstructured data search. This paper articulates the lessons learned from these applications that have guided the design of the Auto-Pipe tool set. What did we do initially? How did that succeed, and how did that fail? What did we do to correct what failed? What remains to be done? Ultimately, the success of a development environment such as Auto-Pipe depends on the benefits that it provides to application developers. Those application developers are primarily interested in the properties of their applications, not the tool set. This is probably the most important lesson that can be learned by a designer of tool sets.

2. APPLICATIONS

In this section, we provide a brief description of five primary applications that have been used to guide the development of of the Auto-Pipe tool set. They are presented in the order in which they were developed, which will be significant in the discussion that follows. There are two points worth noting here. First, each of these applications is substantial in its own right. While several "toy" applications were also constructed during the course of this research, the primary focus has been on real applications that have significant user communities. Second, while they all fall into the class of streaming data applications, there is considerable diversity among the set.

2.1 Approximate Text Search

The text search application includes the following capabilities: exact matching, approximate matching, regular expression matching, and combining operations. In our implementation, the three alternative matching operations are deployed on an FPGA with the combining operations taking place on the general-purpose processor [6].

Exact match. The exact matching operations are based upon Rabin-Karp theory [13]. The algorithm is as follows. The keywords of interest are hashed into positions in a bit vector. Text to be searched is then hashed and the resulting bit vector position is checked for the presence of a keyword. On a hit, there is either a keyword match or a hashing collision. In either event, the hit is delivered from the FPGA to the processor where software determines whether a true positive keyword match or a false positive hashing collision has occurred.

Approximate match. With approximate matching, keywords in a query can be specified with a number, k, of allowed character substitutions or miss-matches. Keywords can be specified to be either case sensitive or case insensitive. Also, individual characters in a keyword can be designated as "don't care" and will match any character.

Regular expression match. The algorithm for regular expression matching operations uses a pipelining strategy that defers state-dependent logic to the last stage, enabling single-cycle state transitions [2]. In addition, a regular expression compiler is used to encode contiguous strings of m input characters and compress the transition table through indirection.

Combining operations. While each of the above search engines has a distinct function, upon a keyword match each returns both the match and match position. Software on the processor is then used to resolve the combining operations including the Boolean operators AND, OR, and NOT as well as proximity operators NEAR and ANDTHEN. The operators AND, OR, and NOT perform their traditional Boolean logic functions at the file level. The operator NEAR is equivalent to AND with the additional constraint that the matching keywords must be within a given distance of one another in the file. The operator ANDTHEN is equivalent to NEAR with the additional constraint that the first keyword must occur earlier in the file than the second keyword.

2.2 Biosequence Search

BLAST, the Basic Local Alignment Search Tool [1], is widely used by molecular biologists to discover relationships among biological (DNA, RNA, and protein) sequences. The BLAST application compares a *query sequence* q to a database D of other sequences, identifying all *subject sequences* $d \in D$ such that q and d have small edit distance between them. The edit distance is weighted to reflect the frequency with which different mutations, or sequence changes, occur over evolutionary time. The BLASTN variant of the application expects both query and database to contain DNA or RNA sequences, and the BLASTP variant expects both query and database to contain protein sequences.

The BLAST application is a critical part of many computational analyses in molecular biology, including recognition of genes in a genome, assignment of biological functions to newly discovered sequences, and clustering large groups of sequences into families of evolutionarily related variants.

BLAST is conceptually a streaming application, composed of a 3-stage pipeline of increasingly expensive but increasingly accurate search operations performed on a database stream. Figure 1 illustrates the basic structure. In stage 1, BLAST detects short exact substrings, or *words*, that are common to both the query and a database sequence, using a hash table of all words in the query. In stage 2, the region surrounding each word is searched to detect pairs of longer substrings that differ by just a few character mismatches. Finally, the small fraction of words that generate such an "ungapped" pair are passed to stage 3, which searches the region around them for pairs of substrings with small edit distance, allowing for substitutions, insertions, and deletions. Only matches that pass this final stage are reported to the user.

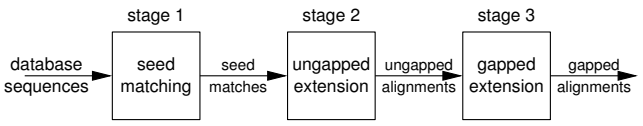

Figure 1: BLAST functional pipeline.

Mercury BLAST [12, 14] accelerates both the BLASTN and BLASTP variants of BLAST using a diverse architecture consisting of both FPGAs and general-purpose processors.

2.3 Triple-DES Encryption

Encryption involves transforming unsecured information into coded information under control of a key. The Data Encryption Standard (DES) operates on 64-bit data blocks

using a 56-bit key. Triple-DES uses three pipelined DES stages to increase the key size. Each stage performs a standard DES encryption using the first, second and third 56-bit keys (64-bits with parity) respectively, and results in a more effective key length of 168 bits (versus 56 bits in DES).

Each of the three pipeline stages takes a single 64-bit input and generates a single 64-bit output. The 56-bit key is handled as a parameter since it is only set once. The performance of our group's Triple-DES implementation using a variety of compute resources (including both processors and FPGAs) is described in [7].

2.4 Gamma-Ray Astronomy

A common experiment in high-energy astrophysics is the examination and characterization of gamma rays generated by extraterrestrial sources. Astrophysicists believe these sources may include pulsars, supernovae, neutron star collisions, and supermassive black holes in galactic nuclei.

The event parametrization task is a computationally intensive step in the ground-based detection of stellar gamma-ray sources. Gamma rays striking the atmosphere result in showers of thousands of photons called Cherenkov radiation. In astrophysics experiments such as HESS [11] and VERITAS [21], these photons are reflected by large (10–17 m) mirrors onto arrays of hundreds of photomultiplier tubes. The photomultiplier tubes transduce the Cherenkov photons, along with the unwanted diffuse background light, to high-voltage analog waveforms. These waveforms are then recorded by fast analog-to-digital converters. We concentrate on the signal processing that is performed on the digitized waveforms to improve the signal-to-noise ratio of Cherenkov photons above background light, and the image processing that characterizes the resulting images to discover features indicative of gamma rays and other cosmic rays.

The topology of the application is depicted in Figure 2. It consists of a configurable number, N, of signal processing pipelines which process each of the digitized waveforms from the 499 photomultiplier tubes. The `Front` section inputs raw pixel data and distributes them to N parallel pipes where the bulk of the computationally intensive digital signal processing is performed. Data from the pipes is merged into the `Back` section, which combines the processed pixel data and performs image-level processing [20].

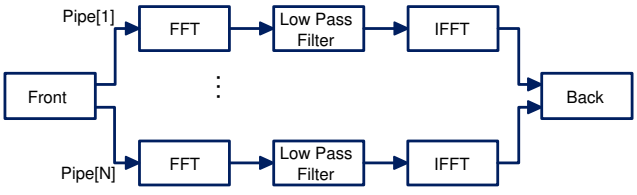

Figure 2: Gamma-ray astronomy algorithm.

2.5 Financial Monte Carlo Simulation

An important application in computational finance is the calculation of value at risk (VAR). The VAR is an indicator of the risk associated with a portfolio of financial instruments. It is defined as the maximum loss that is not exceeded with a given probability over a specified period of time. The probability is specified as a confidence level. The two confidence levels frequently used in practice are 95% and 99%. For example, a VAR of $10,000 at 95% confidence level indicates that the probability that the losses will exceed $10,000 is less than 0.05.

The VAR is calculated by estimating the value of the portfolio at the end of the specified time period. Since the underlying models for pricing financial instruments are driven by stochastic processes, at the end of the time period we obtain a distribution for the value of the portfolio. We use the standard Black-Scholes model for the dynamics of the price of the financial instruments, namely stocks [9].

The Monte Carlo approach to VAR calculation involves simulation of the value of the portfolio at the end of the time period. The differences between the value of the current portfolio and the simulated future portfolios provide estimates of the profit and loss (P&L) over the time period. The VAR then is simply the appropriate value of the sorted P&L estimates. To simulate the values of the components of a portfolio under the Black-Scholes model we need to generate correlated Gaussian random numbers and propagate them forward under the model. The VAR can then be calculated from the resulting distribution. Figure 3 shows the functional pipeline for this simulation. Additional details on the implementation are provided in [18].

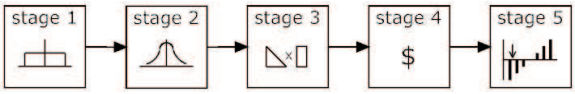

Figure 3: Computation pipeline for financial Monte Carlo simulation.

The pipeline stages are as follows:

- Stage 1: Uniform pseudo-random number generation – the Mersenne Twister is used to generate random numbers that are uniformly distributed between 0 and MAXINT ($2^{32} - 1$).
- Stage 2: The uniformly distributed random numbers are transformed into a Gaussian (normal) distribution with $\mu = 0$ and $\sigma^2 = 1$.
- Stage 3: The vector of independent normally distributed random numbers is transformed into a vector of correlated random numbers. This is accomplished by multiplying the vector by a lower triangular matrix. This lower triangular matrix is obtained by the Cholesky factorization of the specified correlation matrix.
- Stage 4: The correlated Gaussian random numbers are used to generate random walks according to the Black-Scholes model. The values of the portfolio and the P&L values are also calculated in this stage.
- Stage 5: The P&L values are aggregated and sorted to obtain the VAR.

With the exception of stage 5, each of the above stages can be executed in a data parallel manner. We deployed the application above to a set of processor cores, an FPGA, and a GPU using a combination of Auto-Pipe and CUDA. The highest performing mapping is illustrated in Figure 4.

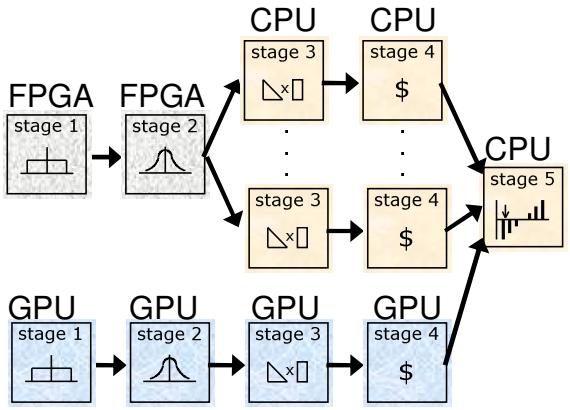

Figure 4: Financial Monte Carlo simulation deployed on 8 Opteron processor cores, a Xilinx Virtex-4 FPGA, and an Nvidia GTX 260 GPU.

3. AUTO-PIPE TOOL SET

Auto-Pipe is a performance-oriented development environment for architecturally diverse systems. Its focus is on applications that are represented as dataflow graphs, and it is especially useful in dealing with streaming applications placed on pipelined architectures. In Auto-Pipe, applications are expressed in the X language [7] as acyclic dataflow graphs. In these graphs, individual computational tasks called *blocks* are connected with interconnections called *edges* indicating the type and flow of data between blocks. Figures 1 through 4 are all examples of this type of graph.

The actual *implementations* of the blocks are written in various languages for any subset of the available platforms (e.g., C for general-purpose processors, HDL for FPGAs, CUDA for graphics engines). Auto-Pipe provides an extensible infrastructure for supporting a wide variety of computation and interconnection devices, simulators, and native languages.

The Auto-Pipe tool set includes an X language compiler, called X-Com [7], the X-Sim federated simulation environment [8], and the X-Dep deployment tool [4]. These components are the basis of the archetypical Auto-Pipe design flow depicted in Figure 5.

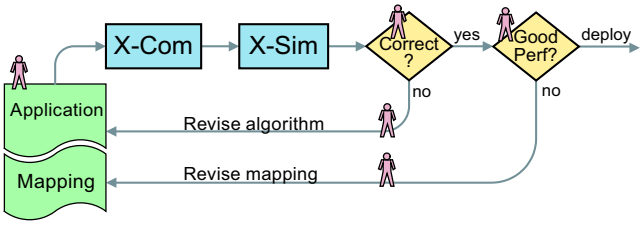

Figure 5: Design flow under Auto-Pipe.

In the Auto-Pipe design flow, X-Com performs compilation of the user-provided application code, supplemented with library code to perform execution profiling, inter-block connections, and high-performance inter-resource communications. X-Sim provides both functional simulation to determine application correctness and performance simulation to profile individual components of the application. X-Dep deploys the complete application to the hardware resources described in the mapping.

Currently, X-Com, X-Sim, and X-Dep are operational and support a variety of computation platforms including native execution on chip multiprocessors, hardware deployment on FPGAs, and simulation of HDL-composed hardware in ModelSim. Extension to graphics engines is currently underway. Processor resources support communication over shared memory or TCP/IP; FPGAs support communication over a PCI-X bus; and all resources support a file-based simulation interconnect used by X-Sim.

One of the primary benefits of the use of Auto-Pipe by an application developer is the automatic instantiation of the communication links between blocks. When constructing block implementations, the programmer must conform to an API that supports input ports for incoming edges and output ports for outgoing edges. The data on an edge is typed, and it is the responsibility of the run time infrastructure to deliver data from a block's output port to the downstream block's input port. When interconnecting blocks, the developer is encouraged to think of edges as having infinite queueing, and appropriate backpressure signals are in place for when the real finite queues are full.

Given a library of existing blocks, each having implementations that can be deployed on a variety of computing platforms, it is then possible to create applications entirely (or at least mostly) at the coordination level, in the X language, by simply specifying the topology of the application. These applications can then be deployed across a range of architecturally diverse systems.

4. LESSONS LEARNED

In this section, we will describe the interdependent relationship between the Auto-Pipe tool set and the collection of applications described earlier. This discussion will proceed in the order in which the applications were originally developed; the same order in which they were introduced.

The text search application and the biosequence search application both pre-dated the Auto-Pipe tool set entirely. During the development of these applications, our group designed and implemented the DMA engine that moves data between the processor's memory and the FPGA (across the PCI-X bus) in a streaming manner. In the text search application data flows from the disk subsystem to the FPGA, which executes the match engines. Hits detected by the match engines flow from the FPGA to the processor to compute the combining operations. In BLAST, input data comes from the disk subsystem to the input edge shown in Figure 1. While Figure 1 illustrates BLAST as a streaming application, the publicly available implementation is not constructed in that manner. As part of our implementation, the design was refactored to correspond to the streaming data model.

The common pattern we observed in these first two applications was the inadequacy of the synchronous data flow model [16] that is commonly assumed in streaming languages. Synchronous data flow makes the assumption that the ratio of data out from a block to the data into a block is known at compile time. This assumption is well founded for many applications, such as signal processing applications, but does not hold when a block acts as a data-dependent filter. In the early pipeline stages of both the text search application and BLAST, when a data value is input to a

block, that block might or might not generate an output data value.

As a result, the initial design of the Auto-Pipe compiler and run time infrastructure made no assumptions about the volume of data that will cross any edge. Since no guarantees are made about the latency of data delivery between blocks, the run time infrastructure can aggregate a buffer of data elements prior to delivery downstream. This is particularly important when moving data across a communication infrastructure like a PCI-X bus, where the overheads of a DMA transfer can then be amortized over a larger set of data.

On the positive side, the experience of developing the first two applications convinced our group that the streaming data computation model is generally good. Conceptually the streaming data organization is well suited to architecturally diverse systems.

As a guiding principle, we decided to construct the Auto-Pipe tool set with the minimal set of functionality we thought sufficient to enable applications to function. As we discovered capabilities that were missing in the tools but were needed for us to build our applications, only then would those capabilities be considered for addition into the tool set. Guided by this philosophy, the initial design of Auto-Pipe only supported streaming data pipes without any built-in signaling capability.

The Triple-DES application was the first substantial application actually developed using the Auto-Pipe tool set. This application taught us the need to carefully consider the approach used for run time parameters. The Auto-Pipe system includes configuration parameters as part of the definition of individual blocks; however, the values for these parameters are set at compile time. For an application like Triple-DES, the cryptographic keys are run time parameters. As such, they also might change during the course of the run, for example, if the user has encrypted a file with one key but wants to encrypt another file with a second key, all part of a single execution.

Our initial approach to handling run time parameters is to provide them to a block via an additional input port. By convention, we draw application diagrams with the primary data flow from left to right and the run time parameterization inputs from the top or bottom. The resulting Triple-DES application is shown in Figure 6.

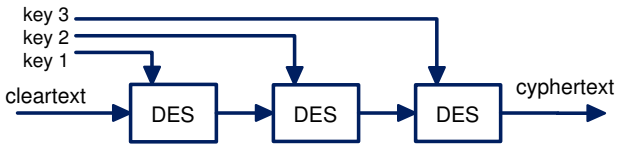

Figure 6: Triple-DES Auto-Pipe design.

The above solution leaves open the question of when should the block update its parameterization from the input parameter port. Since there is arbitrary buffering on each edge, the block has no guarantees as to the relative order of data being made available at its input ports. We will return to this issue later. As a short-term workaround, the Triple-DES application was altered to add a second parameterization input, which told the size of the data set to be encrypted. The block could then read a key, read a data size, and then read, encrypt, and output the specified quantity of data from the primary input port. The clear downside of this approach is that the block must be told the input data size prior to processing any of its input, a requirement that doesn't mesh well with the filters present in our first two applications.

The next application constructed was the gamma-ray astronomy application. The primary lesson that was learned with this application is the need to carefully consider the approach to communications performance. As initially specified, the API that supports data ingest from input ports and data egress to output ports delivers a single data element for each invocation. If the data element is a single data value (or short vector), the overhead associated with the API is a significant fraction of the total time required to deliver data.

Upon completion of the Triple-DES and gamma-ray astronomy applications (which happened fairly close together in time), a change in the specification of the communication API was adopted. This change was prompted primarily by the two issues identified above, and adds both signaling and variable size data delivery to the communication infrastructure.

The signaling capability supports the insertion of a control signal in-band in an edge that connects two Auto-Pipe blocks. A signal that is sent from an upstream block's output port has two guarantees associated with it. First, it triggers the communications run time system to flush the channel to the downstream block's input port. This ensures that no data remains for an indefinite time in intermediate queues or communication channels. Second, it guarantees that the signal will be delivered to the downstream block's input port in sequence. That is, the last data element sent prior to the signal will be delivered ahead of the signal and the first data element sent after the signal will be delivered after the signal. This ordering is not assured, for example, between two parallel channels, even if they connect two common blocks.

The variable size data delivery capability simply allows the block implementation to specify (up to a compile time limit) the number of data elements delivered for each API call. This allows the block to amortize any call/return overheads over a larger set of data.

The fifth major application is the first to use both an FPGA and a graphics engine as application accelerators. As the Auto-Pipe tool set does not yet fully support graphics engines, Auto-Pipe was used for the FPGA and processor subsystem and CUDA was used for the graphics engine. One of the goals associated with this application was to determine how to appropriately extend Auto-Pipe to fully support graphics engines, with the application serving as a point design to test ideas.

As the development of the application progressed, one version deployed stages 1 to 4 on the graphics engine and stage 5 on the general-purpose processor. In this configuration, 15% of the execution time on the graphics engine was measured to be stages 1 and 2. As a result, when we ported stages 1 and 2 to the FPGA, we anticipated a roughly 15% performance gain. Instead, what resulted was a 25% performance drop, a clearly unexpected result. Upon further investigation, it was discovered that as data was being delivered to the graphics engine from the main system memory, the graphics engine was not concurrently executing its computation. We have since succeeded in achieving concurrent communication and computation on the graphics engine under CUDA, and the reintegration of the complete application is currently under way.

Table 1: Application-guided lessons.

Application	Lesson
Text Search and BLAST	synchronous data flow inadequate
Triple-DES	signaling is important
Gamma Ray Astronomy	variable size data delivery needed
Monte Carlo Simulation	need concurrent computation and communications

5. CONCLUSIONS

The Auto-Pipe tool set is aimed at easing the task of developing applications that can execute on diverse architectures, exploiting FPGAs and graphics engines for acceleration. During the course of the development of Auto-Pipe, we have also developed a set of applications that are each individually substantial in their own right. Both the initial capabilities of Auto-Pipe and the first set of substantial enhancements were strongly influenced by what was learned from the applications themselves. These lessons are summarized in Table 1.

This is a pattern that we believe will guide us significantly in the future as well. Rather than add functionality to the tool set that we think someone might want, we will continue to include only functionality that is needed by one or more real applications.

6. ACKNOWLEDGMENTS

This research has been supported in part by NSF grants 0427794, 0751212, and 0905368, NIH grant R42HG003225 (through BECS Technology, Inc.), and Exegy, Inc. R.D. Chamberlain is a principal in BECS Technology, Inc., and Exegy, Inc. M.A. Franklin is a principal in Exegy, Inc.

7. REFERENCES

[1] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucl. Acids Res.*, 25(17):3389–3402, Sept. 1997.

[2] B. C. Brodie, R. K. Cytron, and D. E. Taylor. An architecture for high-throughput regular-expression pattern matching. In *Proc. of 33rd Int'l Symp. on Computer Architecture*, June 2006.

[3] I. Buck, T. Foley, D. Horn, J. Sugerman, and K. Fatahalian. Brook for GPUs: Stream computing on graphics hardware. *ACM Trans. on Graphics*, 23(3):777–786, Aug. 2004.

[4] R. D. Chamberlain, E. J. Tyson, S. Gayen, M. A. Franklin, J. Buhler, P. Crowley, and J. Buckley. Application development on hybrid systems. In *Proc. of ACM/IEEE Supercomputing Conf.*, Nov. 2007.

[5] A. Das, W. J. Dally, and P. Mattson. Compiling for stream processing. In *Proc. of Int'l Conf. on Parallel Architecture and Compilation Techniques*, pages 33–42, Sept. 2006.

[6] M. A. Franklin, R. D. Chamberlain, M. Henrichs, B. Shands, and J. White. An architecture for fast processing of large unstructured data sets. In *Proc. of IEEE 22nd Int'l Conf. on Computer Design*, pages 280–287, Oct. 2004.

[7] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Proc. of Int'l Parallel and Distributed Processing Symp.*, Apr. 2006.

[8] S. Gayen, E. J. Tyson, M. A. Franklin, and R. D. Chamberlain. A federated simulation environment for hybrid systems. In *Proc. of 21st Int'l Workshop on Principles of Advanced and Distributed Simulation*, pages 198–207, June 2007.

[9] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer, 2004.

[10] M. B. Gokhale, J. M. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *Proc. of IEEE Int'l Symp. on FPGAs for Custom Computing Machines*, pages 49–58, 2000.

[11] W. Hofmann, for the H.E.S.S. Collaboration. Status of the high energy stereoscopic system (H.E.S.S.) project. In *Proc. of 27th Int'l Cosmic Ray Conf.*, pages 2785–2788, 2001.

[12] A. Jacob, J. Lancaster, J. Buhler, B. Harris, and R. D. Chamberlain. Mercury BLASTP: Accelerating protein sequence alignment. *ACM Trans. Reconfigurable Technol. Syst.*, 1(2):1–44, June 2008.

[13] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. of Research and Development*, 31(2):249–260, Mar. 1987.

[14] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 49(1):101–121, Oct. 2007.

[15] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[16] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, C-36(1), Jan. 1987.

[17] D. Pellerin and S. Thibault. *Practical FPGA Programming in C*. Prentice Hall, 2005.

[18] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain. Financial Monte Carlo simulation on architecturally diverse systems. In *Proc. of Workshop on High Performance Computational Finance*, Nov. 2008.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of 11th Int'l Conf. on Compiler Construction*, pages 179–196, 2002.

[20] E. J. Tyson, J. Buckley, M. A. Franklin, and R. D. Chamberlain. Acceleration of atmospheric Cherenkov telescope signal processing to real-time speed with the Auto-Pipe design system. *Nuclear Inst. and Methods in Physics Research A*, 585(2):474–479, Oct. 2008.

[21] T. C. Weekes, H. Badran, S. D. Biller, I. Bond, S. Bradbury, J. Buckley, D. Carter-Lewis, M. Catanese, S. Criswell, and W. Cui. VERITAS: the Very Energetic Radiation Imaging Telescope Array System. *Astroparticle Physics*, 17(2):221–243, May 2002.