Exploring New Architectures in Accelerating CFD for Air Force Applications

Jack Dongarra, Shirley Moore,
Gregory Peterson, and Stanimire Tomov
University of Tennessee, Knoxville

Jeff Allred, Vincent Natoli,
and David Richie
Stone Ridge Technology, Bel Air, MD

***Abstract*—[1] Computational Fluid Dynamics (CFD) is an active field of research where the development of faster and more accurate methods is linked to the continuous demand for ever higher computational power. And indeed, for at least two decades, high-performance computing (HPC) programmers have taken for granted that each successive generation of microprocessors would, either immediately or after minor adjustments, make their software run substantially faster. But recent microprocessor design trends including the introduction of multi/many-core designs and the increasingly popular use in HPC of accelerators such as General Purpose Graphics Processing Units (GPGPU) and Field Programmable Gate Arrays (FPGAs), present an unprecedented challenge, namely how to update and enhance the existing large CFD software infrastructure to efficiently use these new architectures. In this paper we address some main issues in this transition and present ideas on using the new architectures to accelerate CFD applications that are of interest to the Air Force. We consider not only multi/many-core but also special purpose (e.g. GPUs) and reconfigurable computing (e.g. FPGAs) architectures. Moreover, we demonstrate benefits of using hybrid combinations where the strengths of each platform can be used to better map algorithm requirements and underlying architecture.**

I. Introduction

Computing technology is currently undergoing a transition driven by power and performance limitations and generational technology advances that provide more and more on-die real estate each year. The past 25 years have seen five orders of magnitude growth in the number of transistors on semiconductor logic devices, and the industry has realized regular performance gains from each technology generation. Steady incremental advances in manufacturing have resulted in shrinking feature size and have allowed more and faster parts roughly every two years. The gains from this development pathway have now saturated and, as is painfully clear, clock speeds have stalled at about 3GHz for the last 4 years. As feature size continues to shrink and the gains realized by increasing core complexity have diminished, microprocessor companies have turned to alternative uses of the on-die silicon. Companies such as Intel and AMD naturally have taken a multi-core approach, i.e. using the additional on-die space as a result of technology advances to place additional instances of their key product, the microprocessor core. The current standard is quad core chips and the development roadmap indicates that 8, 16 and 32 core chips will follow in the coming years. In the past where the market could expect clock doubling every two years it can now expect core-doubling in that period. There is no larger scalar processor under development or even in the planning stage at Intel or AMD. Multi-core is here to stay and will be the predominant mechanism for improved performance for the foreseeable future. The transition is dramatic and significant and will have implications for developers of all types including those in high performance computing science and engineering fields [19], [5]. There is now widespread recognition that performance improvement on CPU-based systems in the near future will primarily come from the use of multi-core platforms.

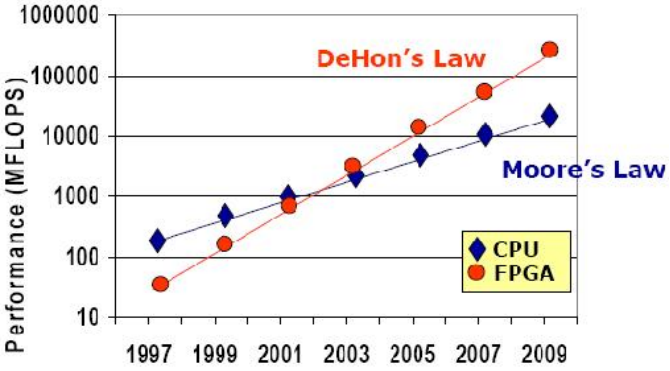

Fig. 1. Moore's *vs* DeHon's law [16].

Populating the new chips with additional x86 architecture microprocessor cores is not the only solution proposed for going forward. IBM, for example, is seizing the opportunity presented by the transition to introduce its own heterogeneous multi-core architecture, the CELL Broadband Engine. Other innovative solutions proposed include GPUs, FPGAs, and ASICs. In fact, trends show that floating point performance on both FPGAs and GPUs has outpaced that of CPUs in recent years. Figure 1 shows data from Underwood et. al. [16] indicating the trends in CPU and FPGA floating point performance over a period of 15 years. Field Programmable Gate Arrays or FPGAs are chips that contain homogeneous programmable circuitry that can be configured and rapidly reconfigured into custom hardware. The scaling of FPGA performance is governed by DeHon's law in a manner similar to Moore's law for CPUs. Dehon demonstrates that FPGA floating point computational density is increasing at a faster

[1]Submitted June 13, DoD HPCMP UGC08, July 14-17, Seattle, Washington

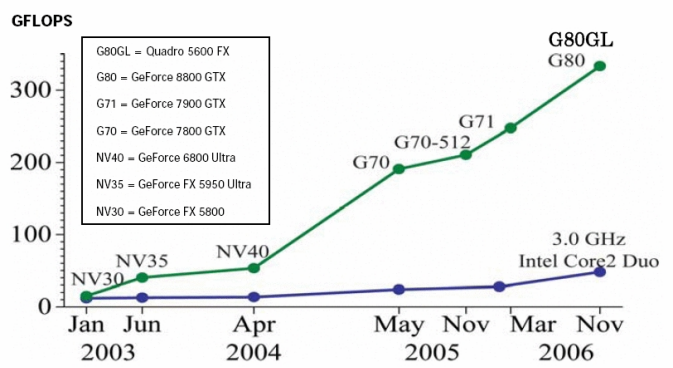

Fig. 2. Floating point performance of CPU *vs* GPU [14].

rate than that of CPUs [9].

GPU companies such as NVIDIA and ATI are leveraging the inherent parallelism of their powerful processors to provide solutions to general computing applications. Figure 2 from the NVIDIA CUDA programming guide shows the recent progress of GPU floating point performance vs. CPU performance. The NVIDIA CUDA development environment provides a traditional and familiar C interface for developers to tap into the GPU SIMD machine.

The problem and the challenge for developers in the new computational landscape is daunting. Developers must produce software for moving targets, some of which do not conform to traditional programming models in any meaningful way. Developers considering next generation software need to understand the inherent parallelism in their problems and then understand the programming model presented by the hardware platform, e.g. multi-core, GPU or FPGA. Developers will need guidance to help map the core algorithmic complexity of their problems to the correct next generation platform. FPGAs excel on problems that can be expressed in moderate levels of task parallelism with significant levels of instruction level parallelism, deep computational pipelines and access to multiple fast independent caches. They are particularly effective on non-floating point problems where their massive reconfigurable fabric can be effectively used for computational processing on integers, fixed point, boolean or custom representations. GPU's are particularly strong with SIMD problems that have significant levels of task parallelism operating on independent data sets. CPU multi-cores are still excellent general purpose platforms that have the most advanced programming model that is most familiar to developers. Processes that have significant branching or are not easily expressed as SIMD methods would map well to CPU multi-cores. Developers are not accustomed to thinking about the strengths and weaknesses of their hardware platforms beyond simple metrics that concern contiguous cache access. In many cases, the optimal solution may well be a hybrid solution combining strengths of each platform. And indeed, as we demonstrate in this paper, some of the CFD acceleration techniques that we address make use of hybrid architectures.

II. CFD for Air Force Applications

Aerospace applications are one of the main drivers and users of CFD technology. CFD simulations are used to predict fluid flows, which in turn are heavily used in Air Force applications such as design of wing and body configurations for aircrafts, design of gas turbine engines, missiles, spacecraft, etc. CFD simulations consist of three main parts, namely (1) a mathematical model for the particular flow being simulated, (2) numerical methods for solving the model, and finally (3) software tools and libraries.

The mathematical model is based on partial differential equations (PDEs). The main PDEs here are the Navier-Stokes (NS) equations [3], which are known to correctly simulate single-phase fluid flows in various regimes. These equations can be simplified for particular flows. For example, compressible inviscid flow corresponds to NS with zero viscosity where the equations are known as Euler equations. Moreover, the NS equations in CFD simulations are often enhanced with supplemental equations to capture various fluid effects. For example, flow with small viscosities and large velocities develops turbulence. Resolving turbulence using just the NS equations (known also as Direct numerical simulation or DNS) requires such a fine mesh resolution that the computational time becomes prohibitively high making the correct calculation impossible even on current supercomputers. Therefore, approximations have been developed to treat turbulence, such as Reynolds-Averaged NS equations (plus turbulence modeling), Large Eddy Simulations (LES), and Detached Eddy Simulations (DES). Other flow effects have spurred up yet more methods. For example, for flows near surfaces there are panel methods and lifting surface methods, convective (i.e. Euler) parts of NS are treated with flux-vector-splitting (FVS) methods, Godunov-type schemes, etc.

The numerical methods used in CFD application encompass various PDE discretization and solution techniques. The main PDE discretization methods are the finite volume method (FVM), the finite element method (FEM), and the finite difference method. All of these standard discretization methods can be enhanced with additional techniques to ensure numerical stability of the solution and at the same time capture discontinuity and other singularity features of the true solution. Some of these techniques are incorporated in the various fluid simulations mentioned above.

By briefly mentioning the various mathematical models and numerical methods used in CFD we want to stress the diversity and complexity of the various computational techniques of interest to Air Force. The field is truly multidisciplinary and over the years a large CFD software infrastructure has been created. In order for our work to have a broader impact in the field our current goal is not to accelerate a specific fluid flow problem, but to concentrate on techniques accelerating certain reoccurring in the field computational kernels. Those are identified in subsection III-A below.

III. Accelerating CFD codes on new architectures

To broaden the impact of our work we concentrate on identification and consequently acceleration of computational kernels that recur in Air Force applications, and that are performance bottlenecks for new architectures. Another useful generalization of this approach is to group and analyze together kernels that share the same pattern of computation and communication. Collela [8] introduced 7 patterns (called "dwarfs"), also described and extended in [1]:

- Dense Linear Algebra
- Sparse Linear Algebra
- Spectral methods
- N-body methods
- Structured grids
- Unstructured grids
- Monte Carlo

All these can be found in Air Force applications and overlap nicely, especially the first two, with our findings.

A. Main computational kernels

We selected the following computational kernels for the focus of our efforts.

- **Sparse Iterative solvers:** Sparse matrices arise in a wide range of computational disciplines, including CFD, where the physics is dominated by local interactions. The solution of $Ax = b$ where A is sparse is a particularly difficult problem for conventional computing architectures. It therefore presents a high value opportunity for acceleration using for example FPGAs.
- **Dense Linear Algebra:** In the situations where sparse matrices do not apply, dense matrices often take their place. When physical interaction are long range or the basis set is extended globally, dense matrices naturally appear. Our current focus here is on one sided factorizations like LU, QR, and Cholesky (and linear solvers based on them). These are of interest and can be accelerated using either multi-core, GPUs, FPGAs, or hybrid combinations.
- **Lattice Boltzmann Method:** LBM is representative of the class of cellular automata methods. Techniques for its solution are also applicable to finite-difference type codes as they have the same pattern of computation and communication. LBM is another method which may present an advantage on reconfigurable hardware over conventional computing architectures because of its inherent simplicity and massive parallelism.

In summary, while the various computational techniques considered to be of interest to the Air Force are very diverse and complex, their bottlenecks are frequently reduced to solving a sparse linear system (e.g. discretization of PDEs with FEM, finite volume, or implicit finite differences methods), dense linear system (e.g. boundary element methods in panel and lifting surface CFD methods, spectral methods, Schur-complement in domain decomposition, subspace diagonalization for eigensolvers or subspace minimization in iterative solvers, etc.), or to a scheme of repeatedly updating certain cells' state based on rules (e.g. cellular automata, explicit finite differences, LBM, certain Monte-Carlo type simulations, etc.). Therefore, accelerating these kernels on the new architectures addressed in this paper will have a significant and broad impact on a wide variety of important Air Force applications. Our stress is on dense and sparse linear algebra kernels as they form the core of most of the Air Force numerical simulations.

B. Some general acceleration ideas

Here we list several general ideas for accelerating kernels for sparse and dense linear algebra. For some of these ideas we have specific numerical results, as shown in Section IV, others are of current research interest and under development.

1) Accelerating algorithms of low CI: The idea here is to try to accelerate algorithms that are not efficient for current architectures (e.g. multi-core). For example, algorithms of low computational intensity (CI: ratio of computations to data required) like Level 1 or 2 BLAS would be bandwidth limited. Indeed, consider for example the Intel Clovertown processor. It is equipped with four cores each capable of a double precision peak performance of 10.64 GFlop/s (or 42.56 GFlop/s for four cores) while the bus bandwidth peak is only 10.64 GB/s, or 1.33 GWords/s. As a result, since one core is largely enough to saturate the bus, using two or more cores on Level 1 or 2 BLAS does not provide any significant benefit. This processor to memory gap is expected to grow by 50% per year according to some estimates.

The same problem is even more important for sparse matrix-vector products due to the irregular memory accesses where performance is often less then 3% of the peak [21]. Reconfigurable computing, with its flexibility in designing parallel memory accesses, presents an opportunity to speed this very large class of problems (see subsection IV-C for a specific example).

2) Accelerating DLA algorithms using hybrid systems: We want to use current advances in accelerating dense linear algebra (DLA) on multi-cores (see subsection IV-A) and achieve further acceleration by using GPUs and FPGAs. In particular, the computations in DLA algorithms can be reorganized by splitting them into tasks that operate on smaller blocks of data, resulting in so-called "tiled" algorithms. The DLA algorithm at hand can then be represented as a Directed Acyclic Graph (DAG) where nodes represent tasks, and edges represent dependencies between them. The execution of the algorithm is performed by asynchronously scheduling the tasks in a way that dependencies are not violated [4]. Scheduling the execution of certain tasks, e.g. so called panel factorizations, as soon as possible may be crucial for the performance as those tasks clear up dependencies for many other tasks that can be executed in parallel, e.g. so called trailing matrix update. Therefore, panel factorization, especially if not overlapped with other work, would be a sequential part of many DLA algorithms, and should be highly optimized. One idea is to have the possibility to accelerate small kernels like panel factorizations using FPGAs, and to accelerate the rich in Level

3 BLAS matrix updates using GPUs (see subsections IV-C and IV-B).

3) Acceleration by developing algorithms of higher CI: This (also very general) idea is to develop new algorithms that compared to current/old algorithms have higher computational intensity. This has been and currently still is an area of extensive research in the field. Examples from the past is the transition from algorithms based on optimized Level 1 BLAS (from the LINPACK and EISPACK libraries) to reorganized DLA algorithms that use block matrix operations in their innermost loops, which actually formed LAPACK's design philosophy. Current examples are work on QR and LU factorizations as their LAPACK implementations have panels involving Level 2 BLAS. For QR, certain out-of-core versions [11] lead to entirely Level 3 BLAS algorithms (see subsection IV-A). For LU a so called randomization technique lead to entirely Level 3 BLAS algorithms (see [2] where we used the technique to develop optimized LU implementation for hybrid CPU-GPU platforms). We point out that algorithms of high computational intensity can be efficiently implemented on the architectures of interest in this paper.

4) Acceleration using mixed precision arithmetic: Finally, as lower precision floating point arithmetic is in general faster than higher precision arithmetic, there is the acceleration idea of mixed precision iterative refinement where the faster lower precision arithmetic is involved for the bulk of the computation, and higher precision only at critical stages, while overall providing results in the higher precision [7], [6], [18]. These techniques are applicable to a wide range of algorithms for solving linear equations and least square problems as well as singular and eigenvalue problems, both dense and sparse matrices (see subsection IV-D).

IV. Some specific examples

A. Acceleration using multi-core architectures

There is a common understanding on how to design certain DLA algorithms for current multicore chips. As mentioned in [4], algorithms should satisfy the following criteria to take advantage of multicore processors:

- fine granularity, as cores are associated with relatively small local memories,
- asynchronicity, to hide the latency of access to memory.

These ideas are applied in current efforts for developing efficient DLA algorithms for multicore [4], [15]. The fine granularity is achieved by splitting the operations into tasks that operate on smaller blocks while asynchronicity is achieved by dynamically scheduling the tasks using a DAG, as already mentioned in subsection III-B2 and as illustrated on Figure 3. On the left we give a typical graph where the nodes in orange represent the sequential parts of an algorithm and in green the tasks that can be done in parallel. The asynchronous dynamic scheduling should be such that the execution of the sequential tasks in orange is overlapped, without violating any dependencies, with the tasks in green. This is done by defining a "critical path", that is the most time-consuming sequence

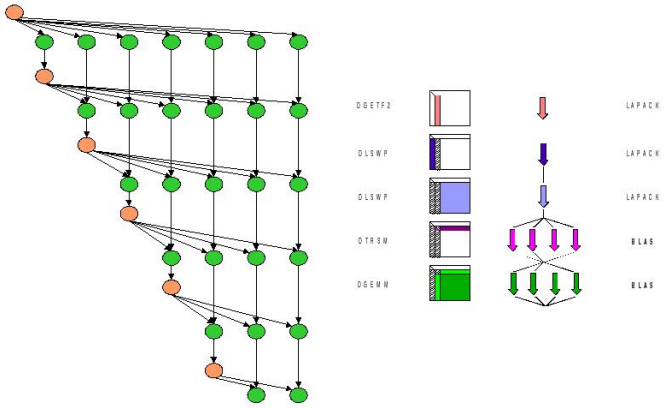

Fig. 3. LU factorization with part of its associated DAG (left) and task splitting showing the sequential panel factorization and parallel trailing matrix update (right).

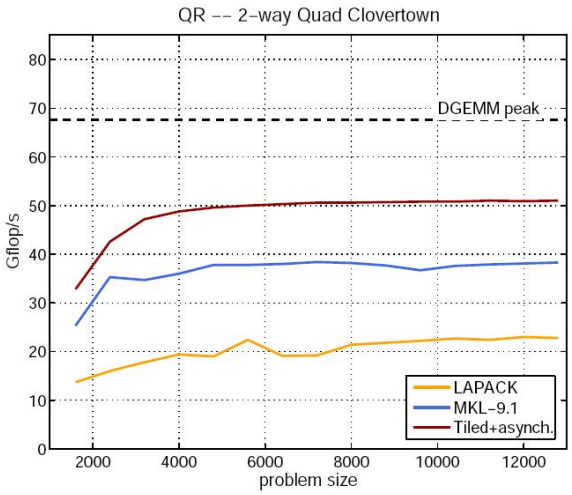

Fig. 4. Performance of a QR factorization (from [4]).

of basic operations that must be carried out sequentially even allowing for all possible parallelism, and scheduling for execution the tasks from the critical path as soon as possible (i.e. when all dependencies have been computed). This ensures that the sequential part of the algorithm will be optimally overlapped with tasks outside the critical path. Note also that no matter the number of cores available an algorithm will be always limited by the time needed to execute its critical path on a single core. As the performance of a single core is already limited this is one more motivation that further speedups may be expected in using hybrid approaches, where the critical path may be executed for example on FPGAs.

Data storage is also essential for effective computations and Block Data Layout [12] can be successfully applied to tiled algorithms. Results of these techniques are encouraging, as shown on Figure 4 for the case of QR factorization [4]. Variations of these ideas can be also recognized in algorithms for GPUs [2], the CELL BE [13], and even FPGAs (e.g. in the case of out-of-core FPGA problems or multi-FPGA use).

B. Acceleration using GPUs

GPUs, especially with the introduction of CUDA[14], are getting much easier to use for general purpose computing. And indeed, there is a very easy and efficient way of using them for DLA, described as follows. First, as GPUs are better suited for data-parallel computations, the tasks splitting has to be done within the BLAS level (BLAS level parallelism). This gives us one obvious way of using them for any LAPACK algorithm, namely by just replacing BLAS calls with BLAS for GPUs (e.g. CUBLAS [2]). In this approach memory is allocated on the GPU, the CPU runs the LAPACK code which is a sequence of BLAS calls that get executed on the GPU. There are no large memory transfers as the matrix to be factored and the work space stay only on the GPU throughout the computation. There are no programming efforts in this approach and we get a good performance for large problems, but still, we are not getting close to the sgemm performance peak (see [2]).

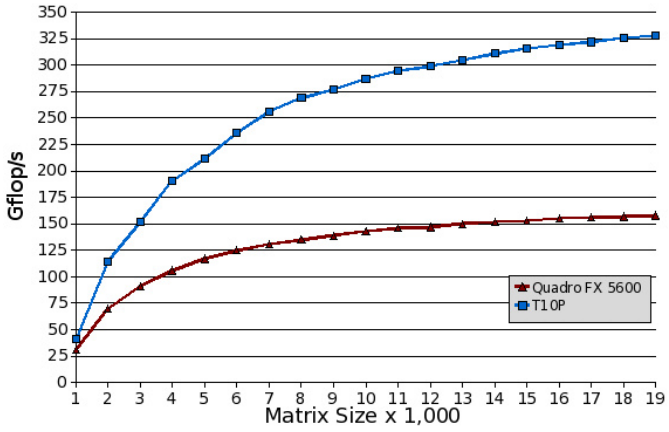

Fig. 5. Performance of Cholesky factorizations using an NVIDIA Quadro FX 5600 GPU in single precision floating point arithmetic.

Similar to designing algorithms for multi-core, we can design GPU algorithms that use asynchronous task execution. An easy way to do it is for hybrid GPU-CPU/FPGA platforms. Let us consider for example the left-looking block Cholesky factorization [10, p. 86]. A step of the algorithm involves two tasks that are independent and can be asynchronously scheduled, namely a "large" sgemm-type update of the trailing matrix can be started on the GPU and at the same time a Cholesky factorization of a small block (from the diagonal of the matrix) on the CPU (using LAPACKs spotf2), resulting in overlapping (or hiding) the sequential small task with the large highly parallel task. Adding this optimization more than doubles overall performance on smaller problems compared to the purely BLAS level parallelism. Figure 5 shows the performance in single precision floating point arithmetic on an NVIDIA Quadro FX 5600 and NVIDIA's next generation T10P card (this card also supports double precision arithmetic). Note that although this approach requires insignificant development efforts, the achieved 328 GFlop/s is very impressive and moreover this algorithms performance has scaled, in the sense that performance has doubled by simply using the new generation card which has double the capabilities of the previous (e.g. number of cores has increased from 120 to 240). More detail can be found in [2].

[2]See http://www.nvidia.com/object/cuda_home.html

C. Acceleration using FPGAs

We already saw several motivating examples for the need for highly optimized small DLA kernels. In particular, these are kernels on the critical path in tiled algorithms designed for multi-core systems, or algorithms for hybrid GPU-CPU/FPGA platforms as shown in the example from subsection IV-B above.

Small kernels indeed can be highly accelerated (compared to single core) using FPGAs. Figure 6 shows the performance of an FPGA LU implementation using floating point arithmetic of various precisions. The blue bars give the LU factorization time on matrices of size 128×128. For more details see [18].

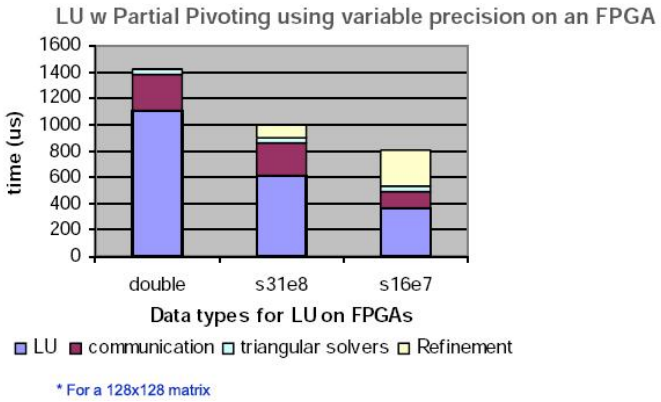

Fig. 6. Performance of LU factorizations on an FPGA enhanced Cray-XD1 in various precision floating point arithmetic [18].

Another example that we want to point out is FPGA designs for sparse matrix-vector multiplication (SMVM). We ported a blocking SMVM algorithm to a Virtex-2 (XC2V3000) FPGA using an assembly implementation. Although this FPGA is two generations behind in current technology the results were encouraging, projecting that a modern Virtex-5 FPGA would boost the performance of our design approximately up to 1.5 GFlop/s. We also want to point out to an earlier design for an Xilinx XC2VP70-7 FPGA where compared to OSKI [20] on a Pentium 4 microprocessor, the FPGA implementation achieves up to $20\times$ speedup [17].

D. Acceleration using mixed precision calculations

Numerical precision on FPGAs can be explicitly designed, however higher precision designs consume a larger amount of resources, and run significantly slower than lower precision designs (see Figure 7).

This allows us to use the mixed precision iterative refinement technique [7], [6], [18] to accelerate FPGA based linear solvers. Figure 6 illustrates a possible speedup on a particular

Characteristics of multiplier on an FPGA* (using DSP48)

Data Formats	DSP48s	Frequency (MHz)	GFLOPs
s52e11 (double)	16/96	237	1.42
s51e11	16/96	238	1.43
s50e11	**9/96**	245	2.61
s34e8	9/96	289	3.08
s33e8	**4/96**	292	7.01
s23e8 (single)	4/96	339	8.14
s17e8	4/96	370	8.88
s16e8	**1/96**	331	31.78
s16e7	1/96	352	33.79
s13e7	1/96	336	**32.26**

* XC4LX160-10

Fig. 7. Performance of various precision floating point multipliers on an XC4LX160-10 FPGA using DSP48 [18].

FPGA. There are limitations to the success of the iterative refinement technique, such as when the conditioning of the problem exceeds the reciprocal of the accuracy of the lower precision computations, in which case the higher precision algorithm should be used. We have studied the technique also on conventional microprocessors for both dense [7] and sparse [6] linear systems.

V. Conclusions and Future Work

Various physical limitations in microprocessor design have resulted in trends that present an enormous challenge to software developers on how to create new or update existing software packages so that they can take advantage of the new and up-coming architectures. In this paper we presented solutions on how to accelerate CFD for Air Force applications not only for multi/many-core but also special purpose and reconfigurable computing architectures. The approach taken was not to concentrate on accelerating a specific CFD Air Force application but to accelerate kernels sharing the same patterns of computation and communication. Several kernels were identified, 4 main ideas on accelerating them were outlined, and specific examples achieving significant speedups on the new architectures of interest here were given. Hybrid architectures proved to be a promising direction in accelerating key classes of problems as it allows us to better map algorithm requirements and underlying architecture.

Our current and future work is concentrated on developing algorithms and libraries related to the 4 main acceleration ideas outlined.

Acknowledgment

Part of this work was supported by the U.S. Air Force Office of Scientific Research under contract # FA9550-07-C-0089, the U.S. National Science Foundation, and the U.S. Department of Energy. We thank NVIDIA and NVIDIA's Professor Partnership Program for their hardware donations.

References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick, *The landscape of parallel computing research: A view from berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] Marc Baboulin, Jack Dongarra, and Stanimire Tomov, *Some issues in dense linear algebra for multicore and special purpose architectures*, Technical Report UT-CS-08-615, University of Tennessee, 2008, LAPACK Working Note 200.

[3] G. K. Batchelor, *Introduction to fluid dynamics*, Cambridge University Press, 1967.

[4] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Technical Report UT-CS-07-600, University of Tennessee, 2007, LAPACK Working Note 191.

[5] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Julien Langou, Piotr Luszczek, and Stanimire Tomov, *The impact of multicore on math software*, the Proceedings of workshop on state-of-the-art in scientific and parallel computing (Para06). Springer's Lecture Notes in Computer Science 4699 (Umeå, Sweden), 2007, pp. 1–10.

[6] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimire Tomov, *Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy*, ACM Transactions on Mathematical Software **34** (2008), no. 4.

[7] Alfredo Buttari, Jack Dongarra, Julie Langou, Julien Langou, Piotr Luszczek, and Jakub Kurzak, *Mixed precision iterative refinement techniques for the solution of dense linear systems*, Int. J. High Perform. Comput. Appl. **21** (2007), no. 4, 457–466.

[8] P. Colella, *Defining software requirements for scientific computing*, 2004, presentation.

[9] André DeHon, *The density advantage of configurable computing*, IEEE Computer **33** (2000), no. 4, 41–49.

[10] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Numerical linear algebra for high-performance computers*, SIAM, 1998.

[11] B. Gunter and R. van de Geijn, *Parallel out-of-core computation and updating of the QR factorization*, ACM Trans. Math. Softw. **31** (2005), no. 1, 60–78.

[12] F. G. Gustavson, *New generalized data structures for matrices lead to a variety of high performance dense linear algebra algorithms*, (June 20-23, 2004), 11–20, In Proceedings of PARA 2004, Workshop on state-of-the art in scientific computing.

[13] Jakub Kurzak and Jack Dongarra, *Implementation of mixed precision in solving systems of linear equations on the cell processor: Research articles*, Concurr. Comput. : Pract. Exper. **19** (2007), no. 10, 1371–1385.

[14] NVIDIA, *NVIDIA CUDA Programming Guide*, 6/23/2007, Version 1.0.

[15] G. Quintana-Orti, E. S. Quintana-Orti, E. Chan, F. G. van Zee, and R. A. van de Geijn, *Programming algorithms-by-blocks for matrix computations on multithreaded architectures*, Technical Report TR-08-04, University of Texas at Austin, 2008, FLAME Working Note 29.

[16] Keith Underwood Sandia, *Fpgas vs. cpus: Trends in peak floating-point*, February 2004.

[17] Junqing Sun, Gregory Peterson, and Olaf Storaasli, *Sparse matrix-vector multiplication design on fpgas*, FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (Washington, DC, USA), IEEE Computer Society, 2007, pp. 349–352.

[18] Junqing Sun, Gregory D. Peterson, and Olaf Storaasli, *High performance mixed-precision linear solver for fpgas*, IEEE Transactions on Computers, to appear.

[19] Herb Sutter, *The free lunch is over: A fundamental turn toward concurrency in software*, Dr. Dobb's Journal **30** (2005), no. 3.

[20] Richard Vuduc, James Demmel, and Katherine Yelick, *Oski: A library of automatically tuned sparse matrix kernels*, Journal of Physics: Conference Series **16** (2005), no. 1, 521+.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, *Optimization of sparse matrix-vector multiplication on emerging multicore platforms*, the Proceedings of workshop on state-of-the-art in scientific and parallel computing (Para06). Springer's Lecture Notes in Computer Science 4699 (Supercomputing), 2007.