Anatomy of a Globally Recursive Embedded LINPACK Benchmark

Jack Dongarra*†‡§, Piotr Luszczek*
*Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996, USA †Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, Tennessee ‡School of Mathematics & School of Computer Science, University of Manchester ¶Research reported here was partially supported by the National Science Foundation and Microsoft Research. Email: {dongarra,luszczek}@eecs.utk.edu

***Abstract*—We present a complete bottom-up implementation of an embedded LINPACK benchmark on the iPad 2. We use a novel formulation of a recursive LU factorization that is recursive and parallel at the global scope. We believe our new algorithm presents an alternative to existing linear algebra parallelization techniques such as master-worker and DAG-based approaches. We show an assembly API that allows us a much higher level of abstraction and provides rapid code development within the confines of a mobile device SDK. We use performance modeling to help with the limitation of the device and the limited access to the device from the development environment not geared for HPC application tuning.**

I. Introduction

The debut of the iPad 2 in the first half of 2011 made widely available a portable device whose Gflop/s per Watt performance may only be matched by GPU accelerators. Moreover, ARM has announced a new processor design, called ARM Cortex-A15, that will increase the performance many-fold and thus bridge the gap with desktop computers. Similarly, the accelerator architectures such as IBM CELL and NVIDIA Fermi that initially shipped with up to an order of magnitude slower double precision units that caught up with their single precision counter parts in the second generation of the hardware – the very situation that we have observed in the transition from iPad to iPad 2.

The focus of the mobile platforms is primarily on graphical presentation and gaming with the sole purpose to extract revenue in a market place that is completely different from the common HPC practices. The domination of proprietary operating systems and Software Development Environments (SDKs) often completely precludes well established optimization techniques such as autotuning due to a complicated app installation that is far removed the familiar command line shell paradigm and requires multiple authorization steps to protect the intellectual property of the hardware, the firmware, and the software of the mobile device. The common remedy is to by pass the vendor protections, a process commonly referred to as *jailbreaking*. However, the compromised development systems cannot supply object files nor binaries for the original system and the end users are unlikely to attempt jailbreaking their devices for the purpose of a single HPC app.

This material is based upon work supported by the National Science Foundation.

The confluence of these factors sets a stage for a drastically different development model. The common tools for compilation and tuning now come from the vendor that is interested in optimizing different kinds of codes than a usual HPC workload. And while it is possible to gain substantial insight from an unencumbered version of the hardware (such as a jailbroken device) by running the standard set of HPC tools, the ultimate destination platform remains almost exclusively outside of the control of the developer.

II. Related Work

Many of the remarks made in the seminal papers on matrix-matrix multiplication [1], [2], [3], [4] are applicable to mobile devices. We used the presented techniques extensively including register and cache blocking. The obvious difference is the update of the tuning parameters to fit the processor we target.

Our use of assembly API is reminiscent of CorePy [5]. It includes a runtime that allows dynamic code generation and abstract coding features. That project was aimed at x86, Power, and the CELL architectures. Our assembly API targets ARM processors exclusively and fits well with hermetic mobile deployment systems with specific development regime. Apple's Xcode 4 offers xcodebuild command which would be the closest to what CorePy requires but it might not be sufficient for all uses. We feel that CorePy would create a perfect match for much more open development boards such as PandaBoard. To summarize, CorePy creates an alternative SDK while our assembly API supplements it.

Due to the throughput of 2 cycles for its double precision fused multiply-add instruction, ARM resembles IBM Cell's even-odd pipeline and, consequently, much may be learned from the work done there [6], [7], [8]. The similarities go further as ARM is used in gaming devices; it still has to ship with viable vendor BLAS; has vector floating-point instructions for single precision only; and there was no proper pipelining for double precision in the first generation of devices.

III. Optimizing Matrix-Matrix Multiply

The majority of the time of the LINPACK benchmark [9] is spent in matrix-matrix multiply routine called DGEMM (Double precision Matrix-Matrix multiply) by BLAS (Basic Linear

Platform	Instruction	Throughput	Latency
Cortex-A8†	Add	9-10	9-10
	Multiply	11-17	11-17
	Multiply-Add	19-26	19-26
Cortex-A9*	Add/Sub	1	4
	Multiply	2	6
	Multiply-Add/Sub	2	9

†Based on revision r1p1 of ARM specification
*Based on revision r2p2 of ARM specification
The actual cycle count depends on the data, eg. the presence of subnormal values, short-circuiting logic of integral values.

TABLE I
ASSEMBLY INSTRUCTION SPECIFICATIONS THAT ARE THE MOST RELEVANT FOR DOUBLE PRECISION MATRIX-MATRIX MULTIPLY.

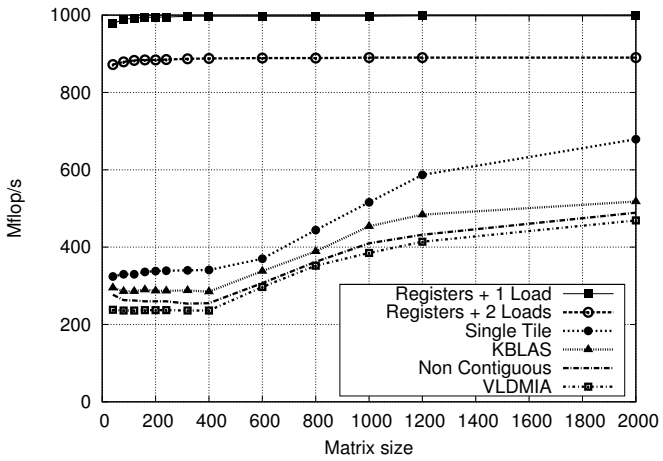

Fig. 1. Performance of multiple matrix-matrix variants on iPad 2 using a single core.

Algebra Subprograms) [10], [11]. A specific variant of this function is commonly referred to as Schur's complement.

Table I shows the two most recent architectures which were featured in Apple's tablets: A4 and A5. The timing specification of the processors comes from ARM documents [12], [13] and closely matches what we could determine with micro-benchmarking. Apple A4 is based on ARM Cortex-A8 and A5 on ARM Cortex-A9. The most important difference from the perspective of a floating-point intensive code is the lack of pipeling of the FPU (Floating Point Unit). The VFPLite (also called VFPv3) coprocessor in A4 does not pipeline either of the double precision instructions. More modern ARM coprocessors do, however, and this includes VFP10 and VFP11 designs – they shorten the latency of add and multiply instructions down to 4 and 5 cycles, respectively.

Another feature worth mentioning is a limited dual-issue capability of both processors. Only the upcoming ARM Cortex-A15 is claimed to include much more superscalar capabilities.

We start by constructing a cache reuse model to maximize the issue rate of floating-point operations. In fact, it is desirable to finish as many such operations in every cycle as is feasible given the number of FPU units. The Schur's complement kernel has a high memory reuse factor that grows linearly with the input matrix sizes. Our model is a constrained optimization problem with discrete parameters:

$$\max_{m+n+k\leq T} \frac{2mnk}{2mnk + (mn + mk + nk)} \tag{1}$$

where m, n, and k are input matrix dimensions and T is the number of registers or the cache size. The model captures the number of loads and the floating-point operations in matrix-matrix multiply for rectangular arguments of sizes m by k, k by n, and m by n:

$$C \leftarrow C - A \times B; \quad A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}. \tag{2}$$

When the fraction given by (1) is maximized, the code performs the most amount of floating point operations per each load operation for the kernel under consideration. The number of floating point registers for the recent ARM processors is 32 and the Level 1 cache is 16 KiB – these are the T values for our model. Our model points to register blocking with $m = 3$, $n = 7$, and $k = 1$ which uses 31 registers. Unfortunately, such register blocking parameters do not fit well with commonly used matrix sizes and the performance advantage over a next best blocking parameters (4, 5, 1) were negligible especially when the effects of cleanup code are considered [1]. Another consideration is the cache line size of 32. It determines the internal register blocking k because the load of the first element from the cache line will put the remaining values in the cache as well. By combining this requirement on internal blocking with the Level 1 cache size we arrived at next level of blocking parameters: 40, 40, 4, which gives us some room to spare in cache for inadvertent misses due to associativity and data misaligment. Optimizations that take into account additional levels of cache were not considered at this time as there is evidence of their lesser importance especially in the context of tile algorithms [14].

We used all the modern C features that allow the compilers to perform aggressive code optimization. These include pointer aliasing restrictions (the restrict keyword), alignment specification, constant blocking and tiling for loops derived from our performance model. A further improvement could potentially be achieved with a more detailed instruction scheduling which may be achieved with a portable C code by using code labels as potential targets of goto statements or empty volatile assembly statements both of which prevent the compiler instruction reordering [15], [16], [17]. Such techniques did not feel natural to us in this context and we chose the ultimate alternative: coding in assembly. In the context of mobile space this is hardly a portability problem as nearly all the devices of interest to us use the ARM architecture. The use of the assembly offers independence of the many variables of involved in the process of taking a optimized code and delivering it to a consumer device.

Figure 1 shows selected variants of DGEMM. The plots marked as Registers + 1 Load and Registers + 2 Loads are shown for reference only because they provide a useful bounds on the achievable performance. The former performs only a single load during computation the latter – 2 loads. The former shows that both C and assembly allow to achieve peak performance of the single core – 1000 Mflop/s – because the curve is the same regardless of the source language: C or assembly. The latter is a more practical bound that shows that

```
func = Function("add2numbers", (("a", Ptr_Int32), ("b", Ptr_Int32)))
a = Pointer(Int32, LoadArg(RegAlloc(Int32), 1) )
b = Pointer(Int32, LoadArg(RegAlloc(Int32), 2) )
AddToReg( a, b )
func.save("add2numbers.c")
```

Fig. 2. Sample function that uses our assembly API for adding two numbers.

```
MultiLoad( [a[0], a[1], a[2], a[3]], A )
A += NB
Load( b[0], B[0+0*NB-NB] )
MultiLoad( [c[0][0], c[1][0], c[2][0], c[3][0]], C[0] )
FuseMulSub( c[0][0], a[0], b[0] )
MoveFromRegToReg( C[0], C[1] )
FuseMulSub( c[1][0], a[1], b[0] )
AddToReg( C[1], NB )
FuseMulSub( c[2][0], a[2], b[0] )
```

Fig. 3. Fragment of DGEMM code on NB by NB matrix tile written in our assembly API.

```
compReg1 = RegAlloc(Float64)
compReg2 = RegAlloc(Float64)
commReg = RegAlloc(Float64)

i = 0
while i < 2:
  i += 1

  # computation
  FuseMulAdd( compReg1, compReg2 )

  # communication
  Load( commReg )

  # swap the buffers
  if i == 1:
    compReg1 = RegAlloc(Float64)
    compReg2 = RegAlloc(Float64)
    commReg = RegAlloc(Float64)
```

Fig. 4. Sample implementation of the double-buffering technique.

adding one extra load deteriorates performance by 10%. The remaining plots operate on memory rather than the register file. Single Tile variant operates on a small 40 by 40 matrix that fits in cache and written with Python code that generates assembly. The KBLAS plot shows performance of general purpose DGEMM blocked for cache [3] and it shows the benefit of using intermediate buffer – the Non Contiguous variant does not use such a buffer is visibly inferior. Finally, we show an assembly version that utilized ARM's multi-load instruction denoted with VLDMIA mnemonic. It is inferior to all the other variants as it stresses the load/store units excessively.If the wrong instruction mix is used the performance may suffer even against a code written in a higher level language. This was one of the motivating factors to develop our own assembly API described in Section IV to quickly develop multiple variants to effectively explore the implementation space.

IV. High-Level Programming in Assembly: Language as API

The development environments for mobile devices are commonly oriented towards non-HPC developer base. The available mobile SDKs (Software Development Kits) integrate all aspects of development including compilation, linking, source code revision control, deployment target selection (hardware device type or software simulator), application signing and provisioning, control of resources such as media files (icons etc.) Under such circumstances, it is nearly impossible to achieve a fully predictable development environment from the perspective of floating-point performance. And automation of the tuning process is yet harder to accomplish. One way to regain predictability at the instruction scheduling level and gain full independence from the compiler is to use the assembly language. This is especially important for an operation such as DGEMM where optimal performance may only be achieved when the floating-point instruction is dispatched every cycle or at least as early as the throughput allows. The downsides of using assembly are well known with low programmer productivity being the most prominent one. We have addressed this by treating assembly instructions as API (Application Programming Interface) calls. There are obvious similarities with compiler intrinsics but for the purposes of our codes they do not provide sufficient functionality because on ARM they only cover single precision floating-point arithmetic. The API was implemented as a Python module. A sample code that uses the functions from the module is shown in Figure 2. The figure shows a very simple function that adds two integers. Upon execution, the code will produce a source code of the function with an inlined assembly that can be used directly in the graphical code editor for the mobile device. Aside for automated code generation, our assembly API offers convenient argument manipulation, symbolic register allocation, and generic instruction names that offer independence from the ARM's mnemonic notations (old ARM mnemonics and UAL – Universal Assembler Language), branch label generation and scoping, tracking register's memory location for automatic store address generation. And obviously the Python code that calls our assembler API may use all the standard Python features as deemed appropriate for performance or readability reasons.

A more evolved example is shown in Figure 3 which shows the use of vector loads of the ARM processor (API call: MultiLoad; mnemonic: VLDMIA) and floating point instructions (API call FuseMulSub; mnemonic: FMAC). It uses the throughput of 2 of the fuse multiply-add instruction: floating-point operations are interleaved with load operations.

A common technique for overlapping computation and communication is double-buffering whereby a portion of memory (or cache or the register file) is used for computing while the rest is communicated. Implementing this technique in assembly may be cumbersome as the same piece of code has to be implemented twice: the second time for the swapped buffers. However, when using our assembly API this may be done with only a single copy of code as the buffer swap is done outside of the code generator. This is shown in Figure 4.

Finally, we also would like to mention an important aspect of assembly API that we used for debugging purposes – an often overlooked aspect of HPC development. Since the API is implemented as a module it is possible to simply import a different module that implements the same API. We use this to switch between the backends of our code generator. The debugging backend produces a portable C code that may be executed on the devices or, to use better debugging tools, on a full-featured desktop computer. The functional equivalence

```
function DGETRFR(M, N, column, A, PIV) {
  if N == 1 {                                          single column, recursion stops
    if IsMine(column) {                                check of column ownership
      idx = IDAMAX(...)                                compute maximum of modulus
      DSCAL(A,1/A(idx))                                scale data
      PIV(column) = idx                                set pivot index
    } else                                             not my column
      while PIV(column) is EMPTY Wait()
        Wait()                                         synchronize: wait for pivot
  } else {
    DGETRFR(M, N/2, column,
      A, PIV)                                          recursive call to factor left half
    WaitForFlag(pivoting)                              waiting for pivoting to finish
    DLASWP(...)                                        pivoting forward
    DTRSM(...)                                         triangular solve
    DGEMM(...)                                         Schur's complement
    DGETRFR(M, N-N/2, column+N/2,
      A, PIV)                                          recursive call to factor right half
    DLASWP(...)                                        pivoting backward
    SignalFlag(pivoting)                               announce pivoting is finished
  }
}
```

Fig. 5. Pseudo-code for the recursive panel factorization.

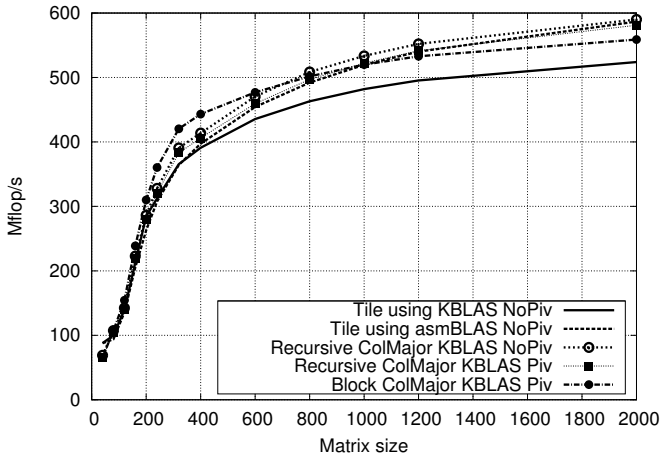

Fig. 6. Performance of various implementations of linear system solve on iPad 2 using a single core.

of the codes generated from the various backends adds a unique aspect to programming in assembly: portability. This is reminiscent of how the iPad simulator works on the OS X desktop machine. The code generated for the simulator is a x86 binary that calls x86 implementations of the iOS APIs thus achieving a functional equivalence between the ARM code for the devices and the desktop binary. Our dual backend approach works well in this context by delivering a C implementation that can work on the simulator and an ARM assembly implementation that works on the device.

V. Recursive LU

The next step is to introduce parallelism in order to take advantage of the dual-core hardware of the ARM processor. The commonly used technique to parallelize dense linear algebra routines is a fork-join model utilized by LAPACK [18] whereby the inherent parallelism of the hardware is abstracted away from the user by parallelising the BLAS routines. By moving the parallelization out of the BLAS, various scheduling approaches [19], [20] have seen a rekindled interest together with DAG-based (Direct Acyclic Graph) runtimes for distributed memory [21], [22], [23].

From the the perspective of the problem at hand, the major problem with any of these parallelization methods is the fact that they require the use of a parameter called *blocking factor*. The parameter determines the granularity of the critical path of the algorithm (the so called *panel factorization*). For the purposes of the small scale of a mobile device, the blocking factor creates an imbalance in the computation that may not be easily hidden when using problem sizes that would fit in the device's main memory. We regard this situation as a form of strong downscaling whereas the system size stays fixed but the problem gets smaller. Consequently, we turn to a method of efficiently implementing LU factorization that does not require a blocking factor. It is often called recursive LU factorization [24], [25] and aside from lack of need for a blocking factor, it has good cache properties and often outperforms other LU implementations. The problem is that the recursive formulation has not seen a parallel implementation in a global scope but rather at the local one with master-worker model being used to distribute computation with a blocking factor for panel factorization [26], [27]. A distributed-memory implementation did not deliver the results that could be used either [28]. Finally, we did not limit the recursion to only a single panel of the matrix and introduce a nested parallelism within the panel while globally using a non-recursive algorithm as was attempted recently [29]. Instead, we have implemented a *new formulation* of LU factorization that is both recursive and parallel in the global scope – a first one of such kind.

Figure 5 shows the pseudocode of our implementation that we call DGETRFR which relates to LAPACK's DGETRF routine that performs the same operation. The implementation is very similar to the original sequential version with the addition of synchronization primitives. In the the terminal case (N==1) the two threads synchronize to make sure that the pivot has been found and successfully applied. This is necessary to ensure that when the recursive call returns both threads can start applying the data computed by the call. In the general case (N>1) the extra synchronization occurs to ensure that the pivoting is finished which is achieved with a single flag. The calls to WaitForFlag() and SignalFlag() are the only two extra required for this. The former is very similar to an atomic compare-and-set operation. Finally, in our implementation we achieve a complete independence of the update calls (DLASWP, DTRSM, DGEMM) due to a cyclic assignment of data to threads. This removes the imbalance caused by the blocking factor.

Figure 6 shows a performance comparison of various implementation of LU factorization by Gaussian elimination in iPad 2 using only a single core. Clearly, the recursive implementation is the fastest (either without pivoting – labeled Recursive ColMajor KBLAS NoPiv, or with pivoting – labeled Recursive ColMajor KBLAS Piv). The block im-

plementation from LAPACK is competitive only for matrix sizes below 500. For reference we also included plot for tile storage versions without pivoting: one using an assembly matrix-matrix multiply kernel (Tile using asmBLAS NoPiv) and with an optimized C kernel (Tile using KBLAS NoPiv). The difference exceeds 10% for most of the matrix sizes which stresses the importance of using assembly code for this kernel.

VI. Management of Concurrency and Parallelism

While Apple's iOS offers a plethora of standard synchronization mechanisms inherited from its OS X lineage we chose to use lock-less synchronization techniques based on cache coherency protocols and memory consistency guarantees. Unlike the approaches [30], [29] that seek to gain performance advantage from more efficient use of cache, we merely sought to have a light weight synchronization operation that would aid our fine grain parallelization method described in Section V. We consider such a method as an alternative to DAG-based approaches [31], [32] especially for fine grain workloads. It encourages synchronization reduction as the algorithm implementer explicitly identifies the events, occurrence of which allow progress of the threads of execution. Such an event-based programming based on consistent memory states appears more natural to us than mutual exclusion guaranteed by mutex locks and signaling through condition variables. In addition, most pthread implementations incur an additional performance penalty in the form of a memory barrier at every call to the thread synchronization primitives.

We did not use atomic operations in our synchronization primitives. In our lock-free synchronization, we heavily rely on shared-memory consistency – a problematic feature from the portability standpoint. To address this issue reliably, we make two basic assumptions about the shared-memory hardware and the software tools. Both of which, to our best knowledge, are satisfied on majority of modern computing platforms. From the hardware perspective, we assume that memory coherency occurs at the cache line granularity. This allows us to rely on global visibility of loads and stores to nearby memory locations. What we need from the compiler is an appropriate handling of C's volatile keyword. This, combined with the use of primitive data types that fit within a single cache line, is sufficient in preventing unintended shared-memory effects.

VII. Performance Results

We present in this section performance results on iPad 2 device from Apple. We believe that the device is powered by an A5 processor that is based on ARM Cortex-A9 design. It features two cores and is clocked at 1 GHz. Unfortunately, we cannot reliably confirm this information as the Apple corporation is free to modify the processor designs licensed from ARM Limited and is believed to have done so by utilizing its various acquisitions such as the purchase of P.A. Semi. Our benchmarking with a register-only matrix-matrix multiply that was presented in Section III confirms the 1 GHz claim and the presence of two cores is clearly visible from the results below.

The runs reported throughout this paper were performed on a device with only the LINPACK app installed. The airplane mode was enabled to remove the influence of networking on our runs. We also explicitly removed any background apps.

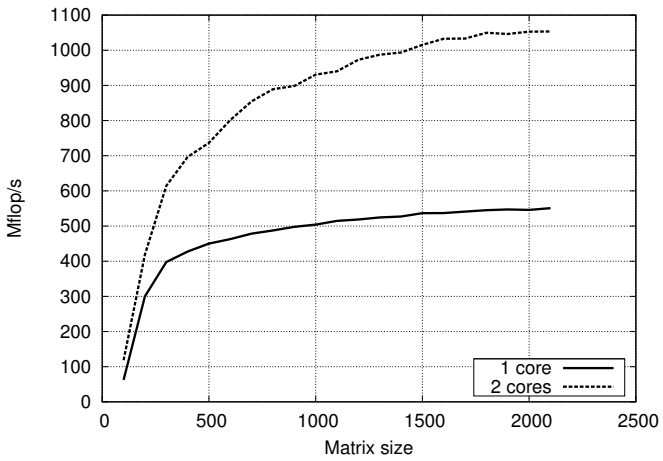

Fig. 7. Performance of linear system solve based on LU factorization by Gaussian elimination on iPad 2.

Device	Performance Gflop/s	TDP Watt	Efficiency Gflop/s per Watt
AMD FireStream[a]	528	225	2.35
NVIDIA Fermi[b]	515.2	225	2.29
AMD Magny-Cours[c]	120	115	1.04
Intel Westmere[d]	96	130	0.74
Intel Atom[e]	6.7	8.5	0.79
ARM Cortex-A9	2	0.5	4

[a]Model 9370
[b]Model M2050
[c]Model 6180SE
[d]Model E7-8870
[e]Model N570

TABLE II
Various performance metrics for a wide range of processors and accelerators.

Figure 7 shows the performance of linear solve through LU factorization. We see a relatively fast increase of parallelism which we attribute to the use of recursion and lack of blocking factor to cause load imbalance impeding parallel performance. We were able to confirm this result on a PandaBoard device where it was also possible to compare directly against the most recent release of ATLAS for ARM – the performance increase is about 10%.

Finally, in Table II we show a different perspective on the performance. We gathered the latest (at the time of writing) double precision performance and power specifications for various processors and devices from multiple vendors. Interestingly, we can observe three-tier division in terms of the Gflop/s per Watt metric. In tier one, with about 1 Gflop/s per Watt, we have desktop and server chips. In tier two, with about 2 Gflop/s per Watt, are accelerators from AMD and NVIDIA. ARM processor is in the third tier, at 4 Gflop/s per Watt.

VIII. Conclusions and Future Work

We have presented a complete implementation of a LINPACK benchmark for iPad 2. Our performance results show that good performance may be achieved by combining an optimized kernels with a novel parallel implementation of the factorization algorithm. For our future work we are looking

into strategies for optimizing other BLAS routines for the ARM processor. Another possibility is the use of an iterative refinement technique [33] which would greatly benefit from NEON engine – ARM's single-precision vector unit.

ACKNOWLEDGEMENTS

The authors would like to thank Clint Whaley for his helpful comments regarding the build and installation of ATLAS on an ARM development board. We would like to also thank Paul Peltz for his help in dealing with Apple's App Store.

REFERENCES

[1] R. C. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *CD-ROM Proceedings of SuperComputing 1998: High Performance Networking and Computing*, 1998.

[2] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing*, vol. 27, no. 1-2, pp. 3–35, 2001.

[3] B. Kågström, P. Ling, and C. V. Loan, "Portable high performance GEMM-based Level 3 BLAS." in *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, 1993, pp. 339–346.

[4] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, May 2008, article 12, 25 pages, DOI 10.1145/1356052.1356053.

[5] C. Mueller, "Synthetic programming: User-directed run-time code synthesis for high performance computing," Ph.D. dissertation, Indiana University, Bloomington Indiana, 2007.

[6] W. Alvaro, J. Kurzak, and J. Dongarra, "Implementing matrix multiplication on the Cell B. E." in *In Scientific Computing with Multicore and Accelerators*. Chapman & Hall/CRC Computational Science series, 2010, iSBN: 978-1439825365.

[7] J. Kurzak, W. Alvaro, and J. Dongarra, "Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor," *Parallel Computing: Systems & Applications*, vol. 35, no. 3, pp. 138–150, 2009, special Issue: Revolutionary Technologies for Acceleration of Emerging Petascale Applications.DOI: 10.1016/j.parco.2008.12.010,.

[8] W. Alvaro, J. Kurzak, and J. Dongarra, "Fast and small short vector simd matrix multiplication kernels for the synergistic processing element of the CELL processor," in *ICCS'08: International Conference on Computational Science*. Kraków, Poland, 2008: Springer, 2008, lecture Notes in Computer Science 5101:935-944, DOI: 10.1007/978-3-540-69384-0_98. Also Tech. Report. UT-CS-08-609 and LAWN 189.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, pp. 1–18, 2003.

[10] J. J. Dongarra, J. D. Croz, I. S. Duff, and S. Hammarling, "Algorithm 679: A set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 1–17, March 1990.

[11] ——, "A set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 18–28, March 1990.

[12] *Cortex™-A8, Revision: r1p1, Technical Reference Manual*, ARM Limited, December 13 2006, issue B.

[13] *VFP11™ Vector Floating-point Coprocessor for ARM1136JF-S processor r1p5, Technical Reference Manual*, ARM Limited, July 6 2007, issue H.

[14] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *Journal of Physics: Conference Series*, vol. 180, 2009.

[15] K. Yotov, K. Pingali, and P. Stodghill, "Automatic measurement of memory hierarchy parameters," *SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 181–192, 2005.

[16] K. Yotov, S. Jackson, T. Steele, K. Pingali, and P. Stodghill, "Automatic measurement of instruction cache capacity," in *Proceedings of the 18th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.

[17] A. X. Duchateau, A. Sidelnik, M. J. Garzarán, and D. A. Padua, "P-ray: A suite of micro-benchmarks for multi-core architectures," in *Proc. 21st Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC'08)*, vol. 5335 of Lecture Notes in Computer Science, Edmonton, Canada, 2008, pp. 187–201.

[18] E. Anderson, Z. Bai, C. Bischof, S. L. Blackford, J. W. Demmel, J. J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. C. Sorensen, *LAPACK User's Guide*, Third ed. Philadelphia: Society for Industrial and Applied Mathematics, 1999.

[19] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, "Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures," in *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, San Diego, CA, USA, June 9-11 2007, pp. 116–125.

[20] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," ICL, Tech. Rep. Technical Report, UT-CS-07-600, September 7 2007, also LAPACK Working Note 191.

[21] J. Poulson, R. van de Geijn, and J. Bennighof, "Parallel algorithms for reducing the generalized hermitian-definite eigenvalue problem," The University of Texas at Austin, Department of Computer Science, Tech. Rep. Technical Report TR-11-05, February 2011, also FLAME Working Note #56.

[22] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemariner, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Distibuted dense numerical linear algebra algorithms on massively parallel architectures: DPLASMA," University of Tennessee Computer Science, Tech. Rep. Technical Report, UT-CS-10-660, September 15 2010.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *PDSEC-11: The 12th International Workshop on Parallel and Distributed Scientific and Engineering Computing*, Anchorage, AK, USA, May 20 2011.

[24] S. Toledo, "Locality of reference in LU decomposition with partial pivoting," *SIAM J. Matrix Anal. Appl.*, vol. 18, no. 4, pp. 1065–1081, October 1997.

[25] F. G. Gustavson, "Recursion leads to automatic variable blocking for dense linear-algebra algorithms." *IBM Journal of Research and Development*, vol. 41, no. 6, pp. 737–755, November 1997.

[26] E. Elmroth and F. G. Gustavson, "New serial and parallel recursive QR factorization algorithms for SMP systems," in *Proceedings of PARA 1998*, 1998.

[27] ——, "Applying recursion to serial and parallel QR factorization leads to better performance," *IBM J. Res. Develop.*, vol. 44, no. 4, pp. 605–624, July 2000.

[28] D. Irony and S. Toledo, "Communication-efficient parallel dense LU using a 3-dimensional approach," in *Proceedings of the 10th SIAM Conference on Parallel Processing for Scientific Computing*, Norfolk, Virginia, USA, March 2001.

[29] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek, "Exploiting fine-grain parallelism in recursive LU factorization," in *ParCo 2011 – International Conference on Parallel Computing*, Ghent, Belgium, August 30-September 2 2011.

[30] A. M. Castaldo and R. C. Whaley, "Scaling LAPACK panel operations using parallel cache assignment," *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 223–232, 2010.

[31] F. Song, A. YarKhan, and J. Dongarra, "Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11, http://doi.acm.org/10.1145/1654059.1654079 DOI: 10.1145/1654059.1654079.

[32] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*, 29 2008-oct. 1 2008, pp. 142 –151.

[33] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy," in *Proceedings of SC06*, Tampa, Florida, Nomveber 11-17 2006, see http://icl.cs.utk.edu/iter-ref.