Abstract—One of the main challenges of using heterogeneous systems results from the need to find the computation-to-hardware assignments that maximize the overall application performance. The important computational factors that must be taken into account include algorithmic complexity, exploitable parallelism, memory bandwidth requirements, and data size. To achieve high performance, a hardware platform is chosen to satisfy the needs of a computation with corresponding architectural features such as clock speed, number of parallel computational units, and memory bandwidth. In this paper five linear algebra computations that are commonly found in compute-intensive applications are selected and evaluated in terms of performance on CPU, GPU, and FPGA platforms across a wide range of data sizes. The results are used to provide guidelines to help select the best performing hardware platform based on the computational factors. Using a cutting edge signal processing application as a case study, we demonstrate the importance of making computation assignments for improved performance. Our experimental results show that a properly implemented heterogeneous system achieves a speedup of up to 39x and 3.8x compared to CPU-only and GPU-only systems respectively.

I. INTRODUCTION

Compute-intensive applications, when implemented in a standard CPU, may not achieve an acceptable level of performance for their required purpose. Some of these, including medical diagnosis[1], weather prediction, or stock and securities market analysis show great potential in their respective fields, but often require alternative hardware solutions such as GPU or FPGA accelerated implementations to operate within an available time budget. Heterogeneous systems combine CPU, GPU, and FPGA architectures into a single system to provide best performance. In order to fully take advantage of heterogeneous systems, some major challenges must be overcome. One of them is choosing the best computation-to-hardware assignment. Another challenge is choosing an implementation for each computation that will efficiently utilize the available resources in each hardware platform. Making a sound design decision can be highly time consuming due to the size of the design space. As such, a method to quickly explore this design space and estimate the performance of an application is needed.

The core of many compute-intensive applications is composed of linear algebra computations such as dot product, matrix-vector multiplication, matrix-matrix multiplication, matrix inverse, and matrix decomposition. In this paper we analyze these five linear algebra computations on CPU, GPU, and FPGA architectures with multiple implementations. Out of the literature [2][3][4][5], successful FPGA architectures for these computations are selected. Multiple implementations for the CPU and GPU architectures from commonly used scientific libraries are used for comparison. These results can be used in systems where all three architectures are available to assist in selecting the best implementation for any computation using a particular matrix size. We present a simple chart showing different architectures and implementations to use in tailoring an application for maximum performance.

The main contributions of this work are:

- Extensive evaluation of the selected linear algebra computations on hardware platforms found in heterogeneous systems.
- Demonstration of the impact of computation-to-hardware assignments on performance.
- A comprehensive chart that guides selection of the proper hardware platform and implementation for each linear algebra computation and data size.
- A case study to evaluate the performance benefits of these guidelines, including data transfer costs.

II. RELATED WORK

Performance of various processing architectures have been evaluated for many computations. CPU, GPU, and FPGA implementations of a Low-Density Parity-Check decoder were compared by Falcao et al.[6] for data sizes 8000x4000 and 1024x512. They concluded that for the smaller data size the FPGA was faster and the GPU was faster at the larger data size. Sotiropoulos et al. designed an FPGA matrix-matrix multiplication architecture[3] and compared its performance to a standard CPU implementation. This comparison was only for specifically sized matrices and did not discuss their CPU implementation. The results showed that the FPGA outperforms the CPU with a speedup of up to 557. A comparison of matrix decomposition by Yang et al.[7] evaluated the performance on CPUs, GPUs, and FPGAs. They analyzed four data sizes from 256 to 1024 and demonstrated that the FPGA was faster than GPU followed by the CPU for both single and double precision floating point. Higher level functions such as 2D filtering by Llamocca et al.[8] and an implementation of Bayesian networks by Fletcher et al.[9] were evaluated using GPU and FPGA architectures. But when making the comparisons, the authors implemented the algorithms solely in one architecture and therefore, chose one particular processor over another. They did not discuss the best implementations for the computations in these higher level functions, only the best implementation for the whole algorithm. Grozea et al. evaluated a sorting algorithm on CPUs, GPUs, and FPGAs[10]...
to speed up the performance of network intrusion detection systems. Their results showed the highest performance architecture was CPU, followed by FPGA and then GPU.

III. ARCHITECTURES AND IMPLEMENTATIONS

The computations that are the target of this work are from two commonly used software libraries: Basic Linear Algebra Subprograms (BLAS) and Linear Algebra Package (LAPACK). Computations from these software libraries have also been implemented in GPUs and FPGAs. The functions used from BLAS include: vector dot product, matrix-vector multiply, and matrix-matrix multiply. From LAPACK we evaluate matrix inversion and matrix decomposition. Since their initial implementations in Fortran, BLAS and LAPACK routines have been used in the AMD C Math Library (ACML) and MathWorks Matlab. Parallel versions of these libraries are also available for GPUs in the form of Compute Unified BLAS (CUBLAS) and Matrix Algebra on GPU and Multicore Architectures (MAGMA). Out of the various FPGA pipelined architectures that have been designed, one was selected [2][3][4][5] for each computation and implemented in Virtex 6 and Virtex 7 FPGA devices.

IV. RESULTS

The five computations were implemented in all three processor architectures using multiple implementations each as shown in Table 1. The FPGA designs were implemented in both devices and execution times recorded. Computations were evaluated using random data for 5x1 to 8000x1 vectors and 5x5 to 8000x8000 square matrices for both single and double precision floating point. The execution time of these experiments are shown in Figure 1 for double precision floating point. The single precision results were similar and not shown due to space constraints. The data transfers are not considered in these execution time. The effect of data transfers will be discussed in Section V. The CUBLAS and MAGMA implementations share the same plot characteristics (color and marker) since only CUBLAS is used for dot product, matrix-vector and matrix-matrix multiply and only MAGMA is used for matrix inverse and Cholesky decomposition. The range of data sizes is very large, however half of the data points are from the 5 to 150 range. This is due to the large amount of overlap of the execution times from various architectures and implementations in this range. For the larger data sizes, the execution times of the various processors are orders of magnitudes different and so a finer granularity in sample points was not required.

A. General Observations

The first characteristic we can highlight for all of the graphs in Figure 1 is that for implementations in both FPGA devices, Virtex 7 (FPGA-1600) always shows slightly better results than Virtex 6 (FPGA-800), which is more noticeable as the size of the data increases. This is mainly due to the higher 1600 Mbps memory bandwidth of the Virtex device. The Virtex 6 device only supports DDR memory bandwidths up to 800 Mbps. Other than this small difference their behaviors and trends are exactly the same. The FPGA implementations generally perform better for smaller data sizes since there is little overhead, thanks to the custom implementation specific to

<table>
<thead>
<tr>
<th>CPU</th>
<th>HW Platform</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Intel Core i7 2600 3.4GHz</td>
<td>AMD C Math Library 5.1.0</td>
</tr>
<tr>
<td></td>
<td>16GB DDR3 @ 1.333Gbps</td>
<td>MathWorks Matlab 2012b 64b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GPU</th>
<th>HW Platform</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nvidia Tesla K20 706MHz</td>
<td>CUBLAS + MAGMA Libraries</td>
</tr>
<tr>
<td></td>
<td>5GB GDDDR5 @ 5.2GHz</td>
<td>MathWorks Matlab 2012b 64b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FPGA</th>
<th>HW Platforms</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Xilinx Virtex 6 LX240T, ML605</td>
<td>[2][3][4][5]</td>
</tr>
<tr>
<td></td>
<td>512MB DDR3 @ 8000Mbp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Xilinx Virtex 7 VX485T, VC707</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1GB DDR3 @ 1600Mbp</td>
<td></td>
</tr>
</tbody>
</table>
each computation. However, the performance of the Cholesky computation on FPGAs is superior to all other architectures due to its ability to process the linear least squares back and forward substitutions in parallel, while still sharing results between them.

The GPU results are very uniform across the range of data sizes. This trend continues up until the very largest data sizes, since an increase in the data size equivalently increases the amount of parallelism for most of the computations. When the GPU architecture has better performance than the other architectures (CPU and FPGA), the amount of parallelism extracted from the computation is enough to fully utilize the large number of hardware resources. However, for Cholesky decomposition the amount of parallelism achievable is much less in the GPU than the other processors due to the design of its architecture. Specifically, the GPU architecture cannot share results as easily across all of the cores. The cost for intercore communication is high since shared data is transferred through global memory. Finally, the implementation also contributes to the amount of exploitable parallelism. For example, the results of the GPU-Matlab implementation tended to be constant across the data sizes, meaning that the operations were more efficiently parallelized than the GPU-CUBLAS implementation.

Reduction operations such as the final addition of the products in the dot product have a reduced amount of parallelism, and given the architecture of the GPU, results in reduced performance. However for the matrix-vector computation, moving from single to double precision uses an increased amount of memory bandwidth and more compute logic resources which reduces the FPGA performance significantly. For matrix-matrix multiplication, the GPU’s ability to process data in parallel and feed all of the cores with data from memory achieves the best performance. The GPU’s architecture is ideally suited for these kinds of computations, with high level of parallelism and where the caching hardware can be fully exploited to remove some of the memory bottleneck. The matrix inverse algorithm has very complex control flow. This leads to very low GPU performance since the path the algorithm takes is control flow dependent, not computation dependent. This computation is not as data dependent and for that reason, both of the FPGAs’ results overlap since the extra bandwidth is not used in the FPGA-1600 configuration. For Cholesky decomposition the FPGA has the best performance. The other implementations are all two and one order of magnitude slower than the FPGA for single and double precision cases respectively (see Figure 1e). It is interesting to note that for Cholesky decomposition, even though the trends are not exactly the same, the results are very similar for implementations using GPU-LA.libs (MAGMA) and GPU-Matlab.

### B. Design Space

The goal of this architecture comparison is to determine the design space for optimal implementation of computations. The chart in Figure 2 shows the architecture with the best performance at any computation and input data size with both single precision (SP) and double precision floating point (DP) using the Virtex 7 FPGA device. For example, the matrix-vector multiply double precision results show that for matrix sizes smaller than 20 the FPGA is the fastest architecture, from 20 to 200 elements the fastest architecture is CPU using ACML, and for matrices larger than 200 using the GPU with Matlab’s implementation is the fastest. As mentioned previously, computations on smaller sized matrices show best performance in the FPGA. The more data dependent operations like dot product and matrix-vector multiply perform better in the CPU using the lower level library ACML. Where there are high levels of parallelism in matrix-vector and matrix-matrix multiplication, the GPU performs the best at larger matrix sizes. We also evaluated this design space using the Virtex 6 FPGA device and found the results very similar to the Virtex 7. The major difference is that the Virtex 6 had reduced performance for dot product, matrix-vector and matrix-matrix multiplication. The other computations were not effected by the reduced capability.

These charts show the effect that factors such as data size, precision, and implementation make on the performance of a computation. Using these charts, programmers can immediately see which computations on particular matrix sizes perform better in different architectures, giving them the ability to tailor the implementations of their algorithms for optimal performance. It also shows that, for applications that have multiple different computations or computations across a wide range of input data sizes, heterogeneous systems will be able to perform better than single architecture systems and clusters. For many data sizes, the performance between the fastest and second fastest implementations was orders of magnitude faster. These differences clearly illustrate the need to optimally assign computations to hardware. The cost of transferring data is very often lower than the associated performance difference.

### V. CASE STUDY

Ultra wideband (UWB) technology is well suited to short-distance, high bandwidth applications such as wireless monitors, "see-through-the-wall" radar imaging and indoor navi-
The heterogeneous system versus just a CPU or GPU. The computing application’s computations are more amenable to a wider system compared to the GPU-only system. This signal processing in the FPGA, resulting in better speedup of the heterogeneous performance of the Cholesky decomposition, which is faster control implied). This algorithm is much more reliant on the 4 shows the speedup versus a GPU-only system (with CPU acting as the intermediary. Figure 3 shows the speedup of the heterogeneous system versus a CPU system for enabling heterogeneous systems to spend time on data transfers and still achieve higher performance than single architecture systems. The guidelines from the chart will be used to create an automatic tool for assignment of computations-to-hardware. Following this, a simulation of the performance of the application in a heterogeneous system will be used to validate the assignments or identify places for improvement. In the future, we will use the results from this paper to design an automatic framework to convert an application from its initial software implementation to a faster implementation in a heterogeneous system.

VI. CONCLUSIONS
In this work we have evaluated the performance of five key linear algebra computations, across a wide range of sizes, and over three different hardware platforms (CPU, GPU and FPGA). The design space charts show which hardware platform should be selected for any combination of computation, data size, and precision. This information can be used to make computation-to-hardware assignments for the bottleneck computations in an application. Then, the remaining computations can be assigned according to their execution and data transfer times to reduce the overall run time of the application. Although many previous works have compared and contrasted CPUs, GPUs, and FPGAs to determine which architecture is better, we show that a single system containing all three architectures results in higher performance. The chart will also indicate if a heterogeneous systems solution is not best for the application. The large differences shown between the fastest and second fastest implementations are key to enabling heterogeneous systems to spend time on data transfers and still achieve higher performance than single architecture systems. The guidelines from the chart will be used to create an automatic tool for assignment of computations-to-hardware. Following this, a simulation of the performance of the application in a heterogeneous system will be used to validate the assignments or identify places for improvement. In the future, we will use the results from this paper to design an automatic framework to convert an application from its initial software implementation to a faster implementation in a heterogeneous system.

REFERENCES