

The best of both worlds: Delivering aggregated performance for high- performance math libraries in accelerated systems

Authors: Dr James Irwin, Mr Simon McIntosh-Smith
Affiliation: ClearSpeed Technology plc.

Abstract

Application accelerators are becoming established as one of the most promising methods of increasing performance in commodity clusters. Today most accelerators are typically used to *replace* the performance of the host CPU for the portion of the application that runs on the accelerator. This approach can be extremely wasteful when multi-core host CPUs are now capable of tens of billions of floating point operations themselves.

In this paper we present an approach that transparently delivers the *additive* performance of the accelerator and host. The methods described have been developed within ClearSpeed's CSXL math library that includes a subset of the Level 3 BLAS and LAPACK libraries.

Introduction

Commodity clusters have rapidly become the de facto standard compute resource in HPC due to their high performance, low cost and familiarity leading to relative ease of use for a growing HPC user community.

As evidence of this revolution, the November 2006 Top500 list contains 341 x86-based systems (68.2%), while five years earlier, the November 2001 list included just 18 such systems (3.6%), a compound annual growth rate of 80%. Today, two out of every three systems in the Top500 are based on x86 CPUs. Of course commodity clusters have their own limitations, which often turn out to be their infrastructure requirements: space, power supply or cooling etc.

Application accelerators are a promising new approach, especially as infrastructure limitations become the roadblock to greater performance. With Gartner predicting that half of all data centres will hit the limits of their power supplies in 2008 [Gartner06], there is already a great deal of interest in accelerators. Accelerators have already started to make a showing in the Top500, with Tokyo Tech's TSUBAME system becoming the first accelerated system to make it into the top ten [TiTech06].

Accelerators are application-optimised processors that can achieve significantly higher performance and power efficiency than general purpose CPUs on specific workloads. The interest in accelerators can be seen in the efforts across academia and industry to use FPGAs, GPUs and game processors such as IBM's Cell for general purpose computation [EETimes07], [GPGPU06], [Cell05]. ClearSpeed's CSX600-based Advance™ X620 accelerator is the first HPC-specific accelerator, and has been designed specifically to meet needs of scientific and commercial HPC applications.

Typically this first wave of accelerators are used in a way that provides *replacement* performance; that is, part of the application runs on the host, and part on the accelerator, but usually not both at the same time. With this approach, at any one time only part of the system's resources is being utilised, the rest lying dormant, its performance wasted. With modern multi-core CPUs now able to deliver low tens of billions of floating point operations (GFLOPS), and accelerators delivering high tens of GFLOPS, we wanted to find an approach that would harness all of this performance simultaneously. In such a scheme the accelerators would be providing *additive* performance, with all of the compute resources working in combination to deliver the maximum performance.

There are potentially many ways in which one could manage the heterogeneous compute resources in an accelerated system. Our aim was to develop an approach that would enable the *transparent* use of both the accelerator and the host at the same time. In order to achieve this, we chose to investigate heterogeneous implementations of the computationally intensive Level 3 BLAS and LAPACK math libraries. These libraries are used in many different applications.

Our goal was to develop heterogeneous versions of key routines from these libraries, such as *DGEMM* from Level 3 BLAS, that could be called like any other library routine, but that would then transparently use the host CPUs and the accelerators in a system. We developed this framework within ClearSpeed's accelerated math library, CSXL.

ClearSpeed accelerators and CSXL



Figure 1 - ClearSpeed's Advance™ X620 accelerator

ClearSpeed's *CSX600* is a purpose-designed, programmable processor for HPC acceleration, using a Single Instruction, Multiple Data (SIMD) array of ninety six 64-bit floating point Processing Elements (PEs) with a peak 64-bit rate of 40.32 GFLOPS [ClearSpeedFPF04]. Two *CSX600s* are mounted on each Advance accelerator, for a peak processing rate of 80.64 GFLOPS double precision; see [CSXArchitecture07] for more detail on the architecture of the *CSX600*. ClearSpeed provides a complete development environment, including an ANSI-C based compiler, a gdb-based debugger, and the ability to debug and profile on the real hardware at run-time.

For the purposes of this paper, we shall focus our attention on the CSXL math library¹ [CSXL07]. This provides a few key Level 3 BLAS and LAPACK routines that have been ported to the Advance accelerator. These routines can be called by an application running on the host, just like any other BLAS/LAPACK library, such as GOTO [Goto02], ATLAS [WhaleyPetitet04], Intel's MKL [MKL] or AMD's ACML [ACML]. Initially, calling a function in CSXL would mean that the function would run on the Advance accelerator if present, otherwise on the host, but not both at the same time.

Our goal was to modify the matrix multiply routine in CSXL, *DGEMM* from Level 3 BLAS, so that it could use both the Advance accelerator and the multi-core host CPUs at the same time, transparently providing the heterogeneous aggregate performance of the system. An Advance accelerator alone can deliver around 66 GFLOPS for *DGEMM*, while a multi-core CPU may also reach tens of GFLOPS of performance.

In the following sections we present our approach to this problem, and the results we obtained. We shall illustrate our findings in the context of the LINPACK benchmark which uses *DGEMM*.

The LINPACK Benchmark

The benchmark used to rank the world's fastest installed machines is well publicised and discussed [Dongarra07]. We shall focus our attention to the component of the benchmark that dominates the running time and the floating-point operations. This is the *update* step following a panel *factorisation* step. Simply put, a portion or *panel* of input data has been processed or *solved* and the results must be carried forward to the remaining unprocessed data or *trailing* matrix. The LINPACK benchmark emits timing information that illustrates the ratio of time spent in the two key components of (panel) factorisation and (trailing matrix) update; for maximum performance, the latter tends to occupy up to ninety percent of the running time.

¹ The version of CSXL used to generate the results in this paper has been superseded, with the latest version (2.50) providing greater performance than that shown here.

In this section we include profiled runs of the LINPACK benchmark running on a workstation using an AMD Opteron 265 (1.8GHz) dual core CPU.

The update step is compute dominated and makes calls to two L3-BLAS [Dongarra90]. Again, the running time of update (*PDUPDATE*) is dominated by spending more than ninety percent of the time in *DGEMM*. This is illustrated in Figure 2.

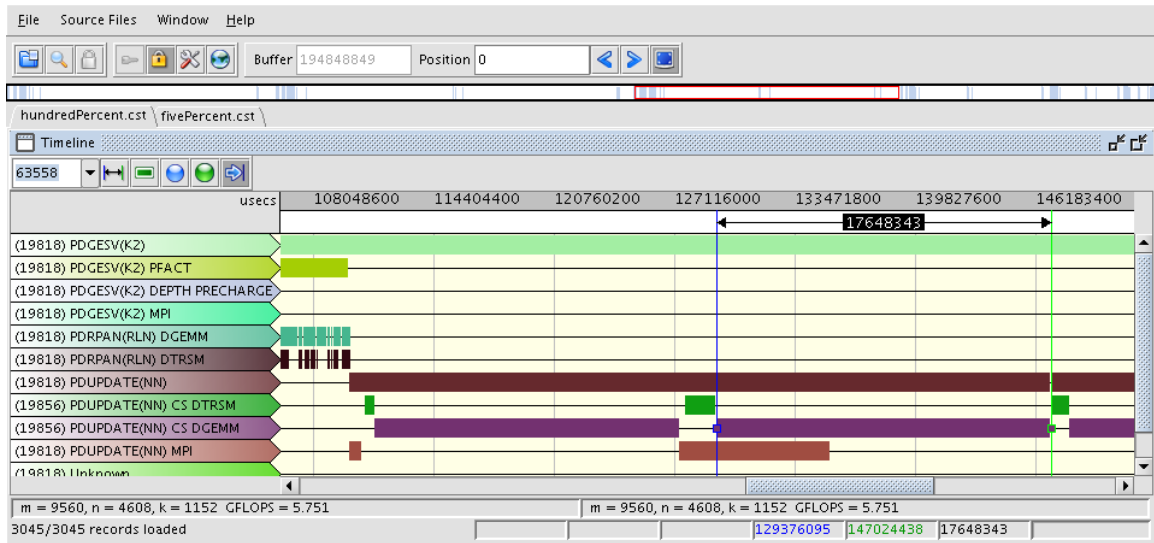


Figure 2 PDUPDATE in LINPACK on a 1.8GHz dual core Opteron only

Here we see a factorisation (*PFACT*) and update (*PDUPDATE*) pair displayed with the ClearSpeed Visual Profiler. This segment of the LINPACK trace is typical and represents the repeated factorise and update pattern that makes up the execution behaviour of LINPACK. The selected *DGEMM* component of the *PDUPDATE* includes instrumentation to record the function parameters, the time taken and a post-processing result to present the rate of execution for the individual step. In this case, the *DGEMM* parameters ($m=9560$, $n=4609$, $k=1152$) and time to execute meant that this function achieved a rate of 5.751 GFLOPS or approximately eighty percent of theoretical peak for the dual-core Opteron in the system being profiled. Empirical efficiency measurements like this will vary between and within runs of LINPACK due to normal system activity.

Replacing the implementation of *DGEMM* so that CSXL performs the computation on the Advance accelerator results in a profile illustrated in Figure 3.

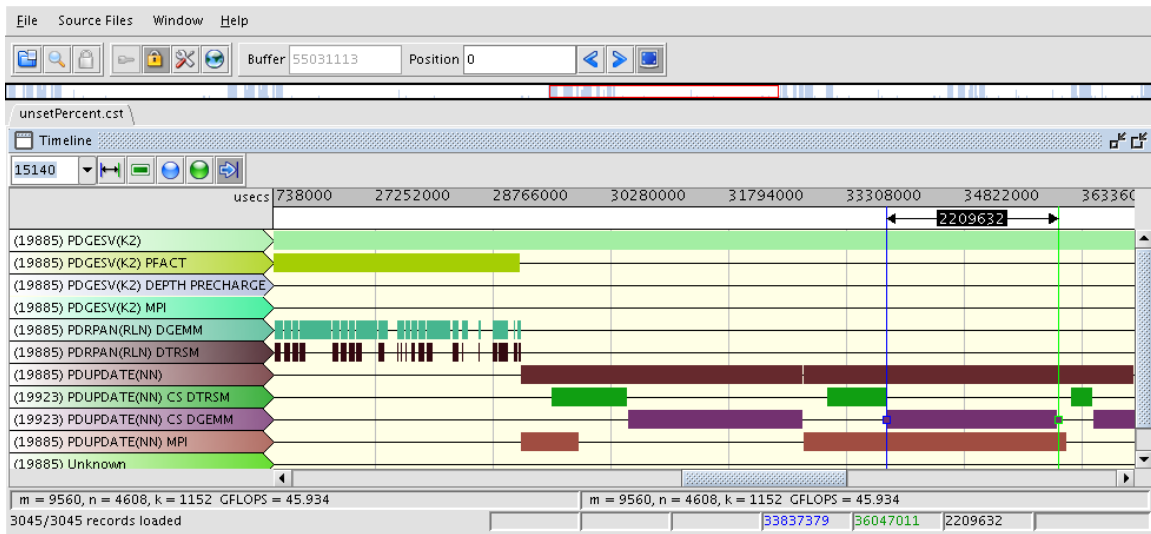


Figure 3 PDUPDATE step in LINPACK using Advance X620 and zero aggregation

Here we see, on a modified scale, the same factorisation and update pair in the LINPACK run for Figure 2. This time, however, the computation for the same *DGEMM* is executed on the Advance accelerator with the CSXL software; all other functions, including the update *DTRSM*, are still being executed on the host. The measured difference between these two figures is the rate at which the *DGEMM* within the update is executed. Here we achieve a *DGEMM* performance of 45.934 GFLOPS, an increase of 699%, significantly reducing the time spent in *DGEMM* and reducing the overall benchmark running time dramatically. The corollary is a much increased benchmark score.

Figure 3 illustrates the combination of host (*x86*) processing for *DTRSM* and so on, and the Advance accelerator for *DGEMM*. This is the most simple and common form of heterogeneous combination of computation.

The running time of LINPACK for medium and large systems remains dominated by *DGEMM* in updating the trailing matrix, and for this we have not yet delivered the full system capability in Figure 3. Simply put, we have only replaced the performance of the host with the accelerator for the update *DGEMM* step. To do better than this, we can combine or aggregate the capability of the Opteron 265 host and the Advance X620 accelerator. We do this with a technique called *host fraction*, where the load is balanced proportional to the capabilities of the host and the accelerator taking into consideration the number of host processors and ClearSpeed accelerators.

Figure 4 shows the result of selecting the optimal balance for this particular system whereby the Opteron is allocated five percent of the floating point operations of the update *DGEMM* step and the accelerator the remainder. We discuss in the next section the detailed mechanisms for load balancing.

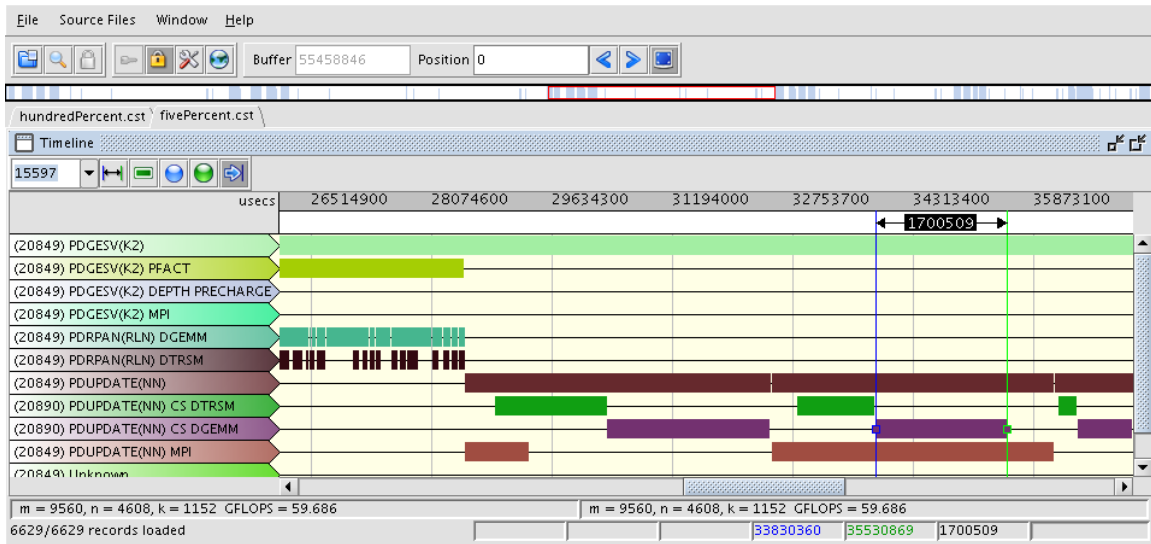


Figure 4 PDUPDATE step in LINPACK using aggregated Advance and Opteron

Again, Figure 4 shows the same factorisation and update pair as in Figure 2 and Figure 3. This time, the rate for computing the update *DGEMM* step is 59.686 GFLOPS, an increase of 928% compared to the host alone, and an increase of 30% over the performance of the accelerator alone. This aggregate *DGEMM* performance is higher than the sum of the two previous component rates, but as the recorded LINPACK score was observed to fluctuate by +/-5%, this score is within the expected performance range. These individual *DGEMM* rates are empirical measurements and are meant to represent generally the effect that can be observed by combining the capability of the host and the accelerators.

Figure 4 illustrates the ability to aggregate the compute capability of the host and accelerator processors for more fine-grained operations than shown in Figure 3. That is, within a single function call, computation was spread between processors from different vendors, with different instruction sets and vastly different architectures.

The following section will discuss how we may balance the load between these heterogeneous processors to optimise for aggregated performance.

Load Balancing

The section above illustrated how we can observe and extract the combined compute capability of the host (*x86*) and accelerator (*CSX600*) processors by splitting the floating-point operations of the L3-BLAS routine *DGEMM*.

Here we shall introduce two methods by which we have approached the problem of balancing the load between the heterogeneous resources. We shall discuss two distinct classes, *Static Host Fraction* and *Dynamic Host Fraction*.

These schemes are discussed in the context of *DGEMM* but are applicable to many data-parallel computations as we shall discuss in a later section.

Static Host Fraction

Simply put, a static scheme for load balancing distributes a constant proportion of the workload to each component of the heterogeneous system. Thus previously, in Figure 4 where the workload ratio was empirically determined to be *1:19* in favour of the accelerator, in a static scheme all *DGEMM* executions would be split in this manner.

The simplicity of this scheme benefits when the workload is well defined and fully characterised and the function has one dominant size and/or shape. We can extend this simple mechanism to provide a vector of *Fractions* for easily identifiable classes such as size or shape. Indeed, the LINPACK benchmark is dominated, in *DGEMM* terms, by a single specific shape of matrix multiplication, and thus a static scheme works well.

The simplicity of this scheme fails to deliver optimal and sometimes even good aggregate performance when the following occur:

- The workload is unknown, *or*
- The workload is not dominated by a single or limited characteristic shape or size, *and*
- The relationship between the performance profiles of the heterogeneous components is not a simple linear scaling.

It is the combination of all three of these points and especially the latter that provide the motivation for a dynamic host fraction scheme, covered in the section below.

We have implemented two simple optimisations to a static host fraction scheme. The first is a simple threshold whereby parameter limits (on the input matrix dimensions, m , n , and k) are used to select a crossover point where load-balancing or the overheads of heterogeneous co-ordination are prohibitive. The second minor optimisation for any load-balancing scheme may consider the smoothness, or lack thereof, of the heterogeneous resources. For example, we may choose to select a distribution of workload that is close to the selected static fraction but that lands at a point of peak efficiency for one or more of the heterogeneous resources. For *DGEMM*, this is at points of the implementation sub-matrix blocking. The difference in performance between peaks and troughs can be large, as much as +/-10 GFLOPS, so this optimisation is important.

The determination of optimal static host fraction is a simple process for a single function parameter set or class. In the instance of LINPACK, the dominant shape has m and n approximately equal and very large and k is NB from the LINPACK parameter file. In the examples above, k is *1152*. As a starting point, we can simply measure the performance of such a *DGEMM* on both host and accelerator processors individually, and then start with this performance ratio as the initial fraction. Empirically, this has resulted

with an optimal host fraction to within one percent (the static host fraction is represented in integer percentage points in CSXL).

Dynamic Host Fraction and Auto-Calibration

Whilst the static host fraction scheme offers implementation and cognitive simplicity, it is both inflexible and inefficient. The static scheme does not consider arbitrary workloads and requires calibration for each and every shape or size that contributes significantly to the behavioural demands of the application. For any given general matrix multiplication shape, the optimal host fraction is dependant on the specific values chosen and trends to a particular value as the parameters are increased.

Here we present a host fraction scheme that replaces a constant by a parametric continuum for load balancing.

Secondly, the implicit workload required to optimise the host fraction for every possible *DGEMM* parameter combination is removed. That is, it is infeasible to capture, process and store the entire parameter space of an optimal host fraction. Instead, we present a mechanism that will efficiently automatically calibrate the analytical load-balancing model on a given system.

We shall be specific with regards to *DGEMM*, although as suggested elsewhere, this technique can be generalised to data parallel computations such as that presented in [IrwinMcIntosh07]. The principal component is an analytical model of the computation. For *DGEMM* we choose the following to model the execution time:

$$d_1mn+d_2mk+d_3nk+d_4mnk$$

Where m , n , and k correspond to *DGEMM* matrix size parameters and the coefficients d_i represent empirical constants that characterise the processor. The first three d coefficients correspond to data management operations, including moving the input and output matrix data from memory to the processor. The fourth d coefficient corresponds to the architectural floating point compute rate for the algorithm excepting data movement costs. This is a model based on the principle components of a *DGEMM* computation and we thus describe it as an analytical model.

This model is reasonably simple in its number of terms and the in the intricacy with which it is able to track the micro-fluctuations in the m , n , and k axes compared to the performance profile of a real system.

The simplicity of the model allows for cheap run-time work-distribution ratio determination. Recall that the optimal empirical host fraction was very near the ratio of the performance of the host and accelerator if they were considered to run the whole function. Thus, we may make a first approximation to the host fraction as the ratio of the model values for the two sets of d coefficients, having one set for the host and one for the accelerator cards. This represents the first simple parametric host fraction scheme.

Refining this dynamic host fraction implementation can include iterative steps to optimise the exact location of the partition, as mentioned before, to select local peak efficiency points. More interestingly, the model may be used to evaluate multiple load balancing options. For example, *DGEMM* floating-point operations may be partitioned along any of the three m , n , or k axes. Since the estimation is cheap, requiring tens of floating point operations all executed on the fast scalar host processor, such optimisation is profitable.

With the model defined, and the application of it determining a parametric host fraction, we come to the calibration of the model, or the determination of the d values that encapsulate the heterogeneous resource components.

For each processor class in the heterogeneous system we derive four d coefficients for the model. We can do this in a number of ways ranging from timing the execution of four *DGEMMs* and a trivial linear algebra process, or by measuring the performance of a large range of *DGEMMs* and more complex linear algebra (least squares fitting for example).

We shall present the results of the most simple calibration technique, whereby four *DGEMMs* are timed to derive the d coefficient values for each processor class in the heterogeneous system. This requires only seconds of machine time and thus makes installation and recalibration inexpensive when the system is upgraded or altered. The d coefficients for the dynamic host fraction are public global environment variables for the implementation we discuss, allowing the user to fine-tune or alter them as they see fit.

Figure 5 presents the overlaid results of the calibrated model and the measured system performance for two systems. Each figure shows the Advance accelerator performance profile (measured and modelled) and the host performance profile. The first system has two AMD Opteron 280s (2.4GHz dual-core) and the second has four AMD Opteron 870s (2GHz dual-core), representing four and eight host cores respectively.

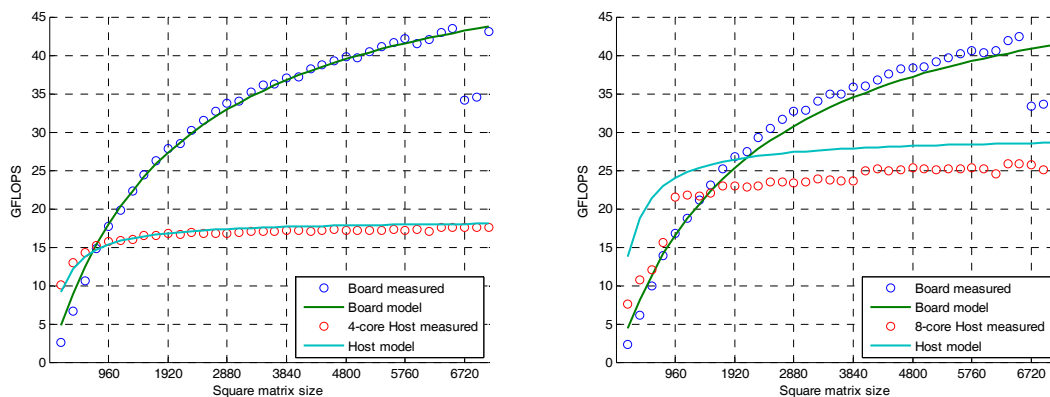


Figure 5 Model evaluation with four and eight core hosts and Advance accelerators

The results shown in Figure 5 illustrate a single auto-calibration attempt and was not iterated to show the best possible results. These then are typical results for contemporary

servers and high-end workstations. The fit for the four-core system is excellent and the fit for the eight-core system is within a reasonable tolerance. The calibration step for both these systems took less than ten seconds using a simple linear algebra process.

The assumptions that permit satisfactory results for a static host fraction are violated by the performance curves in Figure 5. We clearly see a non-linear relationship between the performance curves of the host and accelerator processors. This is for possibly the most simple *DGEMM* parameter profile with $m=n=k$: matrix multiplication of square matrices.

Continuing with these d coefficients, we plot the dynamic host fraction (in percent) for a range of matrix sizes in Figure 6.

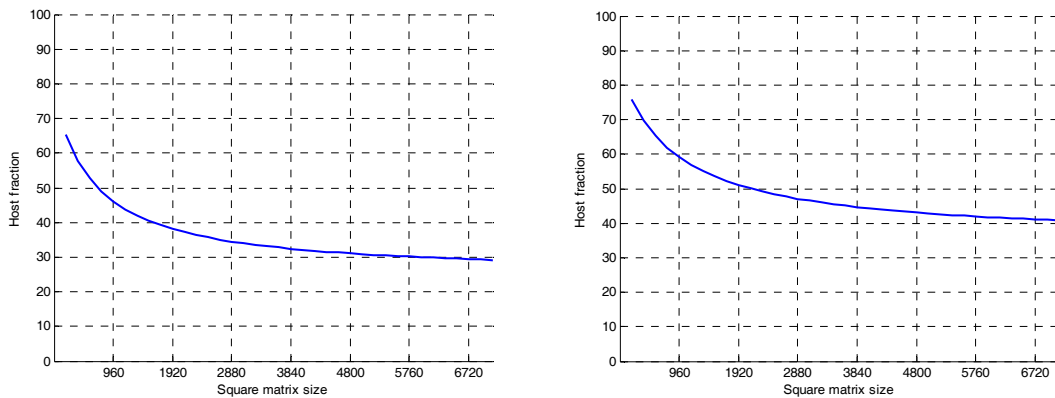


Figure 6 Dynamic host fraction % for 4 and 8 core hosts with Advance accelerators

Applying this host fraction results in the aggregated performance profiles in Figure 7.

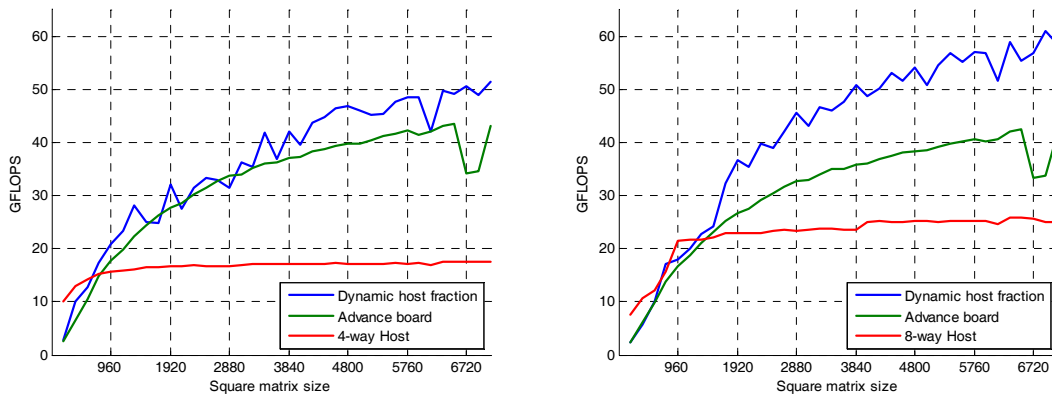


Figure 7 Aggregate performance for 4 and 8 core hosts with Advance accelerators

To measure the efficiency of this scheme, we first estimate what the aggregate system is capable of. We may choose the simple sum of performance for each contributing processor as if they were computing the entire *DGEMM*, or we may sum the component performance estimates for the corresponding fractional *DGEMM*. Figure 8 shows how the simple sum is consistently higher than the fractional component sum. Figure 9

combines the component sum estimate of Figure 8 and the measured performance of Figure 7. The ratio of measured performance to the sum of component performance estimate is plotted in Figure 10.

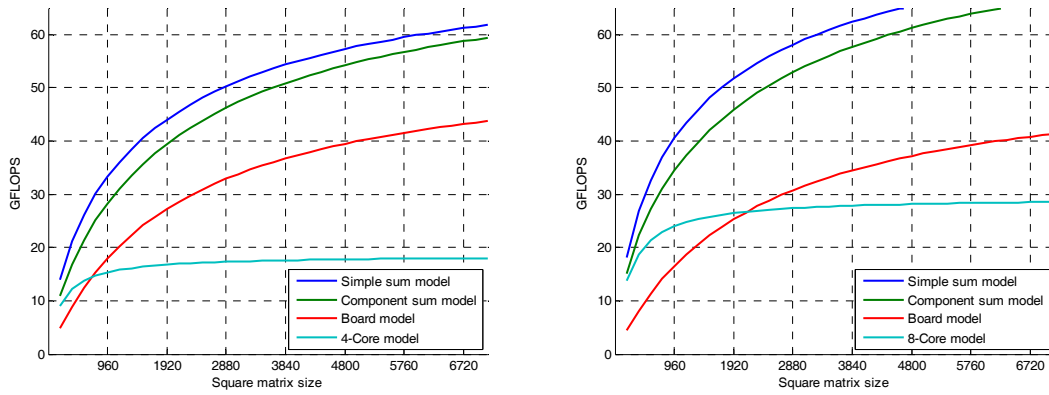


Figure 8 Performance estimates for aggregate systems

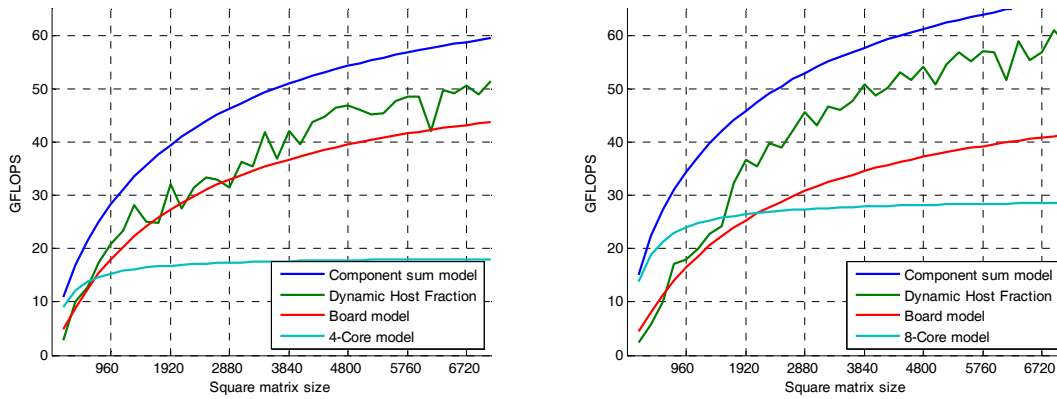


Figure 9 Aggregate performance with estimated target

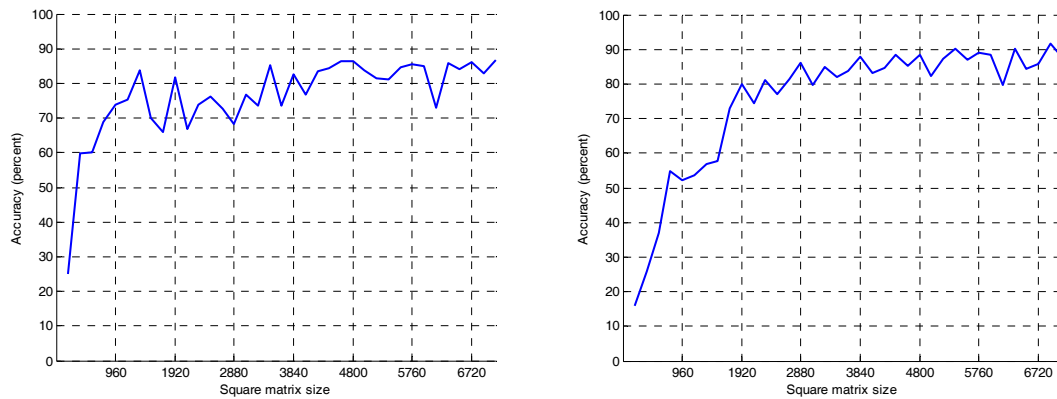


Figure 10 Accuracy of the aggregate performance model

These figures show that for ten seconds worth of automatic calibration, ninety percent of the available performance is delivered. We can increase the complexity of the calibration by increasing the amount of empirical measurement used to solve for the d coefficients. We can also increase the complexity of the analytical model by adding additional terms and so on. These significant expenditures cannot produce much more than ten percent of additional performance. Even in the best of cases, we might only see a fraction of this additional benefit due to overhead costs including run-time load-balancing, and also because of the tracking error between the models and the processor capability at the selected point. Indeed, we have modified the calibration routines to consider the effects that accelerator board management has upon the contribution that the host can make to the aggregated system: running *DGEMM* on an Advance accelerator incurs some overhead on the host as it runs the driver and passes data to and from the accelerator. We found that the modified d coefficients in this experiment were compensated in the intended direction, but these resulted in a worse overall performance. Further analysis showed that the d coefficients for the host had been overcompensated.

We consider this level of efficiency satisfactory to take forward and apply to the LINPACK benchmark as a target application.

LINPACK with Dynamic Host Fraction

We have seen that ten seconds worth of calibration can deliver ninety percent of the estimated aggregate performance with no further effort on the part of the user. Compared to using a manual calibration and static host fraction, this saves hugely in terms of user skill requirements and corollary support demand; it saves in calibration time and the degree of detailed application knowledge too; lastly, and possibly most importantly, this automatic calibration and dynamic host fraction is designed to fit arbitrary applications rather than a specific one.

LINPACK, as discussed previously, is a benchmark heavily dependent on the performance of *DGEMM*. With our focus on tuning and optimising *DGEMM*, we have set a target for dynamic host fraction. We chose a modest problem size of about six gigabytes or about 28,000 unknowns. Additionally, we chose a single workstation rather than a cluster of machines to maintain minimal resource requirements and make the experiments widely relevant and comparable.

We used the same two platforms as in the previous section: a machine with four cores and a machine with eight cores. We calibrated an optimal static host fraction for the update step. This took approximately half a day due in most part to the moderate job size of six gigabytes and to the number of runs required to perform the manual calibration. Ranking runs for the Top 500 list are measured in days to weeks and iterating parameters rapidly consumes otherwise useful or profitable machine time. The resulting benchmark scores were:

- Four 2.4GHz Opteron cores with one Advance accelerator: 41.3 GFLOPS
- Eight 2GHz Opteron cores with one Advance accelerator: 43.6 GFLOPS

The moderate job size means that these scores do not reflect the asymptotic LINPACK performance of the machines.

Using the first auto-calibration values of d for both machines, as used in the previous section, the resulting benchmark scores were:

- Four 2.4GHz Opteron cores with one Advance accelerator: 35.79 GFLOPS
 - 87% of best static score
- Eight 2GHz Opteron cores with one Advance accelerator: 39.57 GFLOPS
 - 91% of best static score

Again, the asymptotic LINPACK performance far exceeded these benchmark scores and would have been more closely reflected in larger problem sizes.

We have investigated manually manipulating the values of the d coefficients and have achieved similar results to the best static scores. This demonstrates that the dynamic scheme is at least not worse than the labour intensive static mechanism. Unfortunately, LINPACK, with its heavy bias to a single shape of *DGEMM*, will not yield a significant positive performance discrimination for this moderately sized job, although it does demonstrate a time-to-90%-of-peak advantage.

It is the time to achieve *most* of what one can reach through extended effort that clearly stands out with this benchmark example. We have converted half a day (or more for significantly larger LINPACK jobs) into ten seconds for 90% of the peak. This reduction in machine-time requirements can be applied instead to productive or revenue generating activity. When machines are upgraded in terms of hardware or software resources, this calibration cost differential is repeated.

Further work

We have suggested that there are numerous points where we might extend or improve the *DGEMM* dynamic host fraction and auto-calibration scheme presented here. We believe that increasing the complexity of the model or the level of analysis for calibration may recover some of the remaining ten percent of performance automatically. We would like, however, to focus this section on generalising this technique to apply to other relevant data parallel algorithms and suggest that these too can fully benefit from efficient aggregation of heterogeneous resources.

Algorithms with simple analytical cost expressions fit this approach. For example, the second major run time function of LINPACK, *DTRSM*, can be modelled in a similar manner to then be partitioned between host and accelerators. Additionally, we may also

decompose *DTRSM* into *DTRSM* and *DGEMM* subcomponents. This permits more flexibility in the heterogeneous exploitation:

- Partition *DTRSM* between host and accelerator, *or*
- Decompose *DTRSM* into two smaller *DTRSM*s and one *DGEMM* and
 - Execute *DTRSM*s on the host and *DGEMM* on the accelerator, *or*
 - Execute one *DTRSM* on each of the host and the accelerator, and partition the *DGEMM* execution between the host and accelerator as discussed in this paper.

Having numerous possible methods for computing *DTRSM* provides significant scope for heterogeneous exploitation. Other L3-BLAS functions offer similar possibilities.

The auto-calibration and dynamic host fraction technique also applies to data parallel computations where the computation being expressed is itself dynamic. Taking an interpreted or dynamically compiled expression as one might for a stochastic computation system [IrwinMcIntosh07] we can calibrate the components, virtual machine instructions or operations in the expression language. From this analogous set of d coefficients, estimated rates of execution for arbitrary expressions can be determined by simple summation, partitioning the total work required between the host and the accelerator.

One final area of further work would be to extend the dynamic host fraction scheme to support multiple heterogeneous accelerators.

Conclusions

In this paper we have demonstrated aggregating heterogeneous compute resources across multi-core general purpose CPUs and accelerators; we combined the host and accelerator and delivered 60 GFLOPS aggregate *DGEMM* performance. We also demonstrated that aggregation can be efficient and flexible; in just a few seconds auto calibration and dynamic host fraction achieved 90% of the best hand-tuned aggregate performance that had taken a whole morning to achieve. We believe that these achievements are widely applicable within HPC.

Acknowledgements

The authors would like to thank their colleagues at ClearSpeed who have been developing the CSXL library and who have significantly contributed to this work, including Matthias Dejaegher, John Gustafson and the entire CSXL team.

References

[Gartner06]: “Gartner Predicts Half of Data Centers Will Run Out of Power by 2008”

<http://www.itjungle.com/tlb/tlb120506-story09.html>

[TiTech06]: “Sun Fire x4600 Cluster, Opteron 2.4/2.6 GHz and ClearSpeed Accelerator, Infiniband”

<http://www.top500.org/system/8216>

[EETimes07]: “FPGA tool bottleneck stalls HPC”

<http://www.eetimes.com/showArticle.jhtml;jsessionid=WH2FVUIOMHM0KQSNDLRSKHSCJUNN2JVN?articleID=197002705>

[GPGPU06]: “SC06 Workshop: General-Purpose GPU Computing: Practice And Experience”

<http://www.gpgpu.org/sc2006/workshop/>

[Cell05]: “A novel SIMD architecture for the Cell heterogeneous chip-multiprocessor”, Michael Gschwind, Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, Takeshi Yamazaki. Hotchips 17, San Jose, CA, August 2005.

http://www.hotchips.org/archives/hc17/2_Mon/HC17.S1/HC17.S1T1.pdf

[ClearSpeedFPF04]: “ClearSpeed’s CSX600: A 50 GFLOP Stream Processor”, McIntosh-Smith, S., Fall Processor Forum, San Jose, CA, October 2004.

[CSXArchitecture07s] “ClearSpeed Whitepaper: CSX Processor Architecture”, February 2007, PN-1110-0702

http://www.clearspeed.com/docs/resources/ClearSpeed_Architecture_Whitepaper_Feb07_v2.pdf

[Goto02]: “On Reducing TLB Misses in Matrix Multiplication”, Kazushige Goto and Robert van de Geijn, FLAME Working Note #9, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55. Nov. 2002

[MKL]: <http://www3.intel.com/cd/software/products/asm-na/eng/307757.htm>

[ACML]: <http://developer.amd.com/acml>

[WhaleyPetitet04]: "Minimizing development and maintenance costs in supporting persistently optimized BLAS", R. Clint Whaley and Antoine Petitet, Software: Practice and Experience", Volume 35, Number 2, Pages 101-121, February 2005.

<http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>

[Dongarra07]: “Performance of Various Computers Using Standard Linear Equations Software”, Jack Dongarra, University of Tennessee, Knoxville TN, 37996, Computer Science Technical Report Number CS - 89 – 85, June 2007

<http://www.netlib.org/benchmark/performance.ps>

[Dongarra90]: “A set of Level 3 Basic Linear Algebra Subprograms”, J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, [ACM Trans. Math. Soft., 16 \(1990\)](#), pp. 1-17.

[CSXL07]: “CSXL User Guide”, version 06-UG-1357 1.13.3.2, February 2007.

http://support.clearspeed.com/resources/documentation/csxl_userguide.pdf

[IrwinMcIntosh07]: In preparation, “Making Accelerators Easy To Use: High-Level Methods For Exploiting The New Generation Of Application-Optimized Accelerators”.

Further Reading

[ClearSpeedIntel06]: “Accelerating the Intel® Math Kernel Library”, Gustafson, J., Greer, B., 2006

http://www.clearspeed.com/docs/resources/ClearSpeed_Intel_Whitepaper_Feb07.pdf