Parametrization of Algorithms and FPGA Accelerators To Predict Performance

Craig P. Steffen
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
csteffen@ncsa.uiuc.edu

July 2, 2007

Abstract

This paper presents a scheme for separately *characterizing computational algorithms and characterizing computing hardware, and then combining those analyses to find the suitability of a piece of hardware for a scientific algorithm. The analysis of the algorithm concentrates on a continuous computational density function,* ρ*, that characterizes the loss of efficiency of computation as a function of local store size.*

A hardware system has multiple layers of cache and data communication, each with a measured bandwidth, latency, and cache size. To predict a limit of the performance of an algorithm on a piece of hardware, each layer is combined with the algorithm's computational density function to compute the limit that layer places on the calculation speed. The lowest calculation speed is then the upper limit of the computation of the algorithm on that hardware platform.

1 Introduction

The flexibility of the fabric of an FPGA is a tremendous asset when performing systematic, repetitive calculations. The functional units on the surface of the FPGA can be allocated and connected in the way that makes most sense for the task at hand. However, the user of an FPGA accelerator pays the price for this flexibility in reduced clock speed, increased difficulty in programming, and loss of memory hierarchy transparency. This paper is concerned with this last point, the requirement of the programmer to explicitly deal with application data transfer.

The details of memory used to be a worry for anyone using a computer[1]. Programmers once directly controlled details of memory access, but modern multi-cache CPUs and compilers have pushed such details outside the concern of day-to-day programming. Data packing and memory use efficiency are still important, but automatic transparent cache controllers mean the scientific programmer only interacts with the caching system in terms of how it slows down their code. However, FPGA processors and other accelerator technologies force programmers to explicitly choreograph data movement in their programs. The fast and efficient cache management logic that application programmers take for granted becomes conspicuous for its absence when someone attempts to program an algorithm on a reconfigurable accelerator.

Programming FPGA accelerators requires that all data movement be understood in the original program and re-choreographed to match the data delivery channels in the target hardware. Designing and programming the re-packaging and movement of data can take time and is not a task that programmers are used to. During, or worse, at the end of this process, the data movement topology may end up significantly slowing down the calculation on that system. Changing the data movement strategy can help resolve this problem, at further increased cost in development time.

The formalism presented in this paper is an attempt to provide a simple, direct way for an domain expert or application programmer to characterize the input data use of an algorithm or a piece of code. The data use information about the algorithm can be compared to characteristics of different hardware architectures to discover what limitations the memory systems of those accelerators will place on the speed of the algorithm. The value of characterizing the algorithm separately from the hardware is that codes are only ported to architectures that support their data movement topology.

The model of this formalism is that the code will be run on an abstract architecture where the bulk of the problem is stored in RAM and the problem is shipped off to a processing unit which has local store spaces successively smaller closer to the processing units. This is a common model for FPGA accelerator systems. However, this formalism is not specifically tied to FPGAs and could be used for dif-

ferent hardware accelerator architectures like the Cell Broadband Engine or General-Purpose Graphics Processing Units (GPGPUs).

Each layer of memory hierarchy has the potential to be the memory transfer bottleneck that slows the algorithm down due to data starvation. This formalism calculates a series of computational speed values, σ_i, one for each layer of the memory hierarchy of a hardware system. The lowest σ value will determine the best that algorithm can do on that hardware architecture. The σ values will be calculated by cross referencing the characterization of the algorithm with the characterizations of a hardware system.

This approach is specifically designed to address the abstract data input needs of the algorithm, unlike more traditional performance analyses[2, 3, 4] which concentrate on the code running on a specific architecture. These assume that the size of the input data stream will be vastly larger than either the output data or input instruction streams, so the output and instructions are ignored in the speed estimations and do not appear in any of the measurements or equations. The speed estimations are entirely based on data input requirements.

The formalism presented here is not designed to predict performance as much as find limitations on that performance imposed by a memory transfer system. It is a code or algorithm pre-analysis to help decide if porting software to an FPGA system is worth the effort.

2 Sample Algorithms

These three algorithms are used below to to illustrate the analysis techniques. They are familiar to any computational scientist and their memory usage characteristics are easily understandable. They are referred to here as algorithms A, B, and C instead of their formal names.

The first“algorithm A”, is a simple dot product of two long vectors, A and B:

```
float A[SIZE],B[SIZE],dot_prod=0.0;
int i;

for(i=0;i<SIZE;i++){
  dot_prod += (A[i]*B[i]);
}
```

The second algorithm, “algorithm B”, is a simple square matrix multiply. The code is a triply-nested loop. It is used to illustrate less trivial data dependencies.

```
float A[SIZE][SIZE],B[SIZE][SIZE];
float C[SIZE][SIZE],temp;
int i,j,k;
for(i=0;i<SIZE;i++){
  for(j=0;j<SIZE;j++){
    float temp=0.0;
    for(k=0;k<SIZE;k++){
      temp += A[i][k]*B[k][j];
      }
    C[i][j] = temp;
    }
  }
}
```

Algorithm C is an NxN interaction problem. This type of problem is common in physical simulations. The assumption is that every data point (particle, atom, etc.) must interact once with every other particle in the simulation. Assuming that the interaction between two particles must be computed only once for the pair, and that the data points do not self-interact, then the total number of point-point computations to calculate for a set of n points is $\frac{n^2}{2} - n$.

```
particle_t particle_list[N_PARTICLES];
int i,j;
for(i=0;i<(N_PARTICLES-1);i++){
  for(j=i+1;j<N_PARTICLES;j++){
    interaction_function(particle_list[i],
                         particle_list[j]);
  }
}
```

3 Characterizing algorithms

This formalism introduces a new technique to characterize a computational algorithm, built around what I will call a characteristic “computational density function”, ρ, of the algorithm. This formalism makes the following assumptions. First, the total input data set consists of m operands each of size s and a total size of $M = m \cdot s$. The total number of computations that must be performed is Z. Each computation requires at least ν operands. The computations will be performed on some arbitrary abstract processor that has a local memory store (analogous to a cache in a real, physical processor). We also assume that there are not memory packing and alignment and access patterns are ideal. While they are are all problems that a programmer must address, they are not considered here.

The definition of a “computation” here may *or may not* correspond to the output of a single physical computational unit. The final answer of this formalism will be computations per second, and so size of a “computation” depends on context. In NAMD[5], for instance, it would probably make sense to define a computation as the calculation of the force on one atom from one other atom. The output computational speed prediction would be in terms of number

of atom-atom forces that could be calculated per second. Even though one such force computation requires several look-ups and elementary operations, the code is built around atom-atom interaction and so that is a logical division.

The computational density function, $\rho(\alpha)$, is the number of computations per byte of data that can be performed by our abstract processor if the local store is of size α. The units of ρ are operations per byte $\left(\frac{ops}{B}\right)$. The model is that the local store will be filled once and all possible computations with just that data performed. When those computations are finished, the processor loads more data out of the total problem M into the local store to continue computing. The density function is a continuous, monotonically increasing function of the local store size (see exceptions two paragraphs down). Generally, the larger the local store, the more computations per byte can be performed before more data is required. Once the local store size α is larger than M, then the entire problem can be computed by loading into the local store once. (This computation density function is being defined as a part of this performance modeling scheme. It is not related to any density function referred to in stochastic modeling of physical systems. The density here is number of computations per byte, similar to the way the MILC[6] collaboration talks about "bytes per flop".)

The computational density function is formulated by calculating the number of operations η possible with α bytes of data and dividing by the size α:

$$\rho(\alpha) = \frac{\eta(\alpha)}{\alpha} \tag{1}$$

The designer of the program designs the η function according to the way the code does calculations based on input data. Finding this function typically requires detailed knowledge of the algorithm. It is often convenient to formulate η in terms of some other independent variable first (such as the number of rows/columns in a square matrix multiply) and then re-parameterize in terms of memory storage α. "Computations" defined in the functions η and ρ are at the convenience of the algorithm designer, so whatever is appropriate for the problem should be used. The units of the final speed predictions will be these computations per second. In NAMD[5], for instance, it might be useful to define one "operation" as the determining of forces between two particles in the simulation. Any smaller definition of operation in that case would not be useful to predicting performance on a large problem.

The computational density function is monotonically increasing unless there are dependencies between performing calculations on different sections of the input stream. If one piece of the calculation must be complete before loading the next set of data (for instance if the first calculation determined which data to load next) then the calculation cannot be streamed and the computational density goes down. Example: for an algorithm, a data block of size S must be stored to do the first calculation. The first such block is A, then second is B and so on. The local store, μ, is large enough to hold A and one half of B. If the calculation of B is independent of that of A, then in one transfer A and half of B can be loaded, thus $\eta(\mu) = 1.5 calculations$. However, if B cannot be loaded until the calculation of A is finished, then the computational density function will get smaller as a function of α because the value of η stays the same:

$$\frac{\eta(1.5\cdot M) = \eta(M)}{1.5\cdot M} < \frac{\eta(M)}{M} : \rho(1.5\cdot M) < \rho(M)$$

The computational density function described here is not unlike the notion of temporal locality discussed[7, 8] by SDSC and Berkeley. However, the temporal locality is a fine grained result measurement, taking into account each memory reference as the code runs. Computational density is *only* concerned with input data to the algorithm, not fetching instructions or out-of-band data. Computational density is also specifically designed to be formulated analytically as a tool to analyze the general memory footprint of an algorithm.

The computational density function generally has a value of $\frac{1}{\nu s}$ when α is just large enough to hold operands for one operation

$$\rho(\alpha = \nu s) = \frac{1}{\nu s},$$

increases with increasing α, and then assumes a constant value when α is larger than M:

$$\rho(\alpha \geq M) = \frac{Z}{M}.$$

Another way to parameterize the computational density function is in terms of a data re-use function, φ:

$$\rho(\alpha) = \frac{Z}{M}\frac{1}{\varphi(\alpha)}. \tag{2}$$

The re-use function can be seen as a re-use *penalty*; that is, if a memory size of α means that all the data must be loaded twice, then the value of φ with α as that memory size will be equal to 2. The higher the penalty function, the lower number of effective computations can be performed per byte of loaded data. The ideal situation is for data to only have to ever to be loaded once, $\varphi = 1$.

An important point to emphasize is this formalism references the general idea of a cache, but it is entirely a characterization *of the algorithm itself* without reference to any hardware. This function can be analytically derived from the code or algorithm without any hardware knowledge.

3.1 Algorithm A: Vector Multiply

The first algorithm is simplest to characterize. Each step of the problem takes two operands and generates one result. Each input operand is used only once, with no opportunity for data re-use. The possible operations function is $\eta = n$ where n is the number of entries available from each vector. Reparameterizing in terms of α:

$$\alpha = 2 \cdot n \cdot s; n = \frac{\alpha}{2s}; \eta(\alpha) = \frac{\alpha}{2s}$$

$$\rho_A(\alpha) = \frac{\eta(\alpha)}{\alpha} = \frac{\left(\frac{\alpha}{2s}\right)}{\alpha}$$

$$\rho_A(\alpha) = \frac{1}{2s} \tag{3}$$

where s is the size of the input operands. A functional graph of ρ_A versus α produces a completely flat graph.

Notice that ρ_A is not a function of the local store size α, and because this is a streaming algorithm, each piece of input data is operated on only once.

3.2 Algorithm B: Matrix Multiply

Matrix multiply has a significant amount of data reuse in the algorithm. The larger the constituent blocks, the more times each matrix value is used. In a square matrix multiply, if the calculation is broken down into sub-blocks, the number of times each value is used is proportional to the size of the sub-blocks.

If we define an multiply-add (or multiply-accumulate) as a single operation, for a square matrix multiply with n by n matrices, the number of operations possible is $\eta_B = n^2(n-1) \approx n^3$ for significantly large values of n. The memory required to store matrices of size n is $\alpha = 2n^2 s$, so

$$n = \sqrt{\frac{\alpha}{2s}}; \eta_B(\alpha) = n^3 = \left(\frac{\alpha}{2s}\right)^{\frac{3}{2}}$$

$$\rho_B(\alpha) = \frac{\eta_B(\alpha)}{\alpha} = \frac{\left(\frac{\alpha}{2s}\right)^{\frac{3}{2}}}{\alpha}$$

$$\rho_B(\alpha) = \frac{\sqrt{\alpha}}{(2s)^{\frac{3}{2}}} \tag{4}$$

3.3 Algorithm C: all-to-all interaction

An all-to-all data topography is typical of physical simulations with action-at-a-distance forces needing to be calculated. As stated above, the total number of particle-to-particle forces to be calculated is $\frac{n^2}{2} - n$.

$$\eta_C \approx \frac{n^2}{2}$$

The amount of storage is proportional to the number of particles to be stored, so finding the computational density function:

$$\alpha = n \cdot s \; ; \; n = \frac{\alpha}{s}$$

$$\eta_C(\alpha) = \frac{n^2}{2} = \frac{\left(\frac{\alpha}{s}\right)^2}{2} = \frac{\alpha^2}{2s^2}$$

$$\rho_C(\alpha) = \frac{\eta_C(\alpha)}{\alpha} = \frac{\left(\frac{\alpha^2}{2s^2}\right)}{\alpha}$$

$$\rho_C(\alpha) = \frac{\alpha}{2s^2} \tag{5}$$

4 Performance Evaluation

Evaluating the limits of the performance of an application uses knowledge of the application as expressed in its computational density function and knowledge of the hardware itself. The hardware is modeled as a series of associated memory size (μ), bandwidth (β), and latency(λ) values. Each layer of a memory hierarchy places a limit on the ability to pass data and thus a limit on the computational speed, or throughput (σ). The slowest value for σ will be the upper limit of the computational throughput of the algorithm on that hardware (imposed by the memory system).

The following equation (derived in appendix A) shows the computation of the speed for a given layer in the memory hierarchy:

$$\sigma(\mu, \beta, \lambda) = \rho(\mu) \cdot \beta \cdot \frac{1}{\left(1 + \frac{\beta\lambda}{\mu}\right)} \tag{6}$$

This equation is based on the premise that the computation is finished when the final input data is transferred into the processor. These estimates ignore the transport of output data. This is done for simplicity's sake and is valid because this method is designed to produce an *upper limit* on performance.

The total time taken to transfer the input data through the layer represented by μ, λ, and β is the sum of all the transfer times. The dimensionless number $\frac{\beta\lambda}{\mu}$ is the fractional impact that the transfer latency has on the total transfer time. Its meaning is more clearly expressed as $\lambda / \frac{\mu}{\beta}$, the latency as a fraction of time time taken to fill up that layer of memory once. In a well-balanced system, this fraction will be much less than one, meaning that layer was designed so that the latency is of minimal impact. In most systems that fraction is small enough to be neglected so the calculation rate equation becomes

$$\sigma = \rho(\mu) \cdot \beta \tag{7}$$

Note in a well-balanced system ($\frac{\beta\lambda}{\mu} \ll 1$) the latency of transfers (λ) has disappeared.

4.1 Performance prediction on the SRC MAP-C

The MAP-C is a reconfigurable computing processor manufactured by SRC Incorporated[9] and installed in the Map-6 reconfigurable computer. The MAP-C consists of two Xilinx Virtex-2 6000 FPGAs and up to 7 banks of 4 MB, 64-bit-wide RAM (called "On-Board Memory", or OBM banks) attached directly to the FPGA. The MAP Processor communicates with the system via a communications module called a SNAP that plugs into a DIMM slot on the motherboard.

The SRC 6/MAP-C system has three layers of communication[10] between the main RAM (where the problem is presumably stored) and the execution units on-board the FPGA. We call the first layer "layer 0", which exists inside the FPGA itself. Transfers within the FPGA happen clock-to-clock, so latency is effectively zero $\lambda_0 \sim 0$. The bandwidth feeding the registers is the "infinite bandwidth"[11] that is discussed in FPGAs.

The second layer of communication is from the OBM to the FPGA block rams. This layer is defined by the FPGAs interface to the OBM banks. The latency is effectively zero (it's only a couple of clocks and it's hidden by the compiler), the total size[12] is $\mu_1 = .6MB$ (roughly the size of all of the block RAM on the FPGA), and the bandwidth is $\beta_1 = 6.4\ GB/s$ (8 bytes per bank per clock from 8 effective banks[1] at 100 MHz).

The next layer of communication is from main RAM through the SNAP to the MAP processor. The bandwidth through the SNAP is $\beta_2 = 1.4\ GB/s$, the transfers are feeding the OBM so $\mu_2 = 28\ MB$, and the latency to start transfers has been measured to be $\lambda_2 \approx 20\mu s$. Transfer latency is not zero, but it is small enough that its effect is amortized over many memory transfers:

$$\frac{\beta_2 \lambda_2}{\mu_2} = \frac{(1.4\ GB/s)(20\mu s)}{(28\ MB)} \approx .001$$

4.2 Algorithm A on MAP-C

Performance prediction is determining how well the characteristic computational density of an algorithm fits within the memory hierarchy of a piece of hardware. We put the characteristic ρ function of the algorithm together with the characteristics of layers of memory hierarchy (each represented by a set of β, μ and λ) and the lowest resulting value of calculation speed, σ, will be the limit of that algorithm on that hardware.

[1]There are 7 physical OBM banks. Six sit between the user algorithm FPGAs and the SNAP. The seventh sits between the user FPGAs and can be read by both of them simultaneously, for an effective 8 simultaneous banks if both FPGAs are participating in the calculations

The computational density of algorithm A is a constant value:

$$\rho_A = \frac{1}{2s}$$

and for the sake of discussion we assume a long vector of single precision floating point values. First we estimate algorithm A on layer 1 of the SRC map

$$\sigma_{A,1} = \rho_A(\mu_1) \cdot \beta_1 = \frac{1}{2s} \cdot \beta_1 = \frac{1}{2(4\ B)} \cdot (6.4\ GB/s)$$

$$\sigma_{A,1} = 0.8\ \frac{G\ ops}{sec} \tag{8}$$

For layer 2 of the Map-C, ρ is a constant, so μ does not enter the equation, the only difference is $\beta_2 = 1.4GB/s$ so

$$\sigma_{A,2} = \frac{1}{2(4\ B)} \cdot (1.4\ GB/s) = 0.46\ \frac{G\ ops}{sec}$$

Given that these operands are single precision floating point values, the transfer from memory will limit this calculation (and any that have the same sort of data access pattern) to .46 GF (single precision). That is unrelated to what kind of operational blocks you can fit into the FPGA.

This analysis is not a complaint about the SRC MAP processor design; the conclusion is that data access topology of this type does not fit well on a MAP processor. When using reconfigurable architectures, we should stress that the this sort of analysis is a crucial first step before attempting to make some piece of code run on an FPGA-based processor (or any other architecture). Before going through the trouble of building an algorithm that works on an architecture one should do the preliminary analysis to determine if it can feed data to the processor such that it is worth doing at all.

In this case of algorithm A, since the operations in question are multiplies, that means that that the highest floating point operational speed possible in this problem would be 1.6 single precision Gigaflops. This is independent of the number of floating point cores that can be instantiated on the Map-C processor. The result of this analysis is that regardless of the programming techniques, this hardware is not a good fit for this algorithm, due principally to insufficient bandwidth into the MAP processor.

4.3 Algorithm B on MAP-C

Algorithm B is much more typical of scientific computational problems; in fact, matrix multiplication is the basis for the popular benchmarking suite High Performance Linpack. The computational density of algorithm B does have depend on the size of the cache,

$$\rho_B(\alpha) = \frac{\sqrt{\alpha}}{(2s)^{\frac{3}{2}}}$$

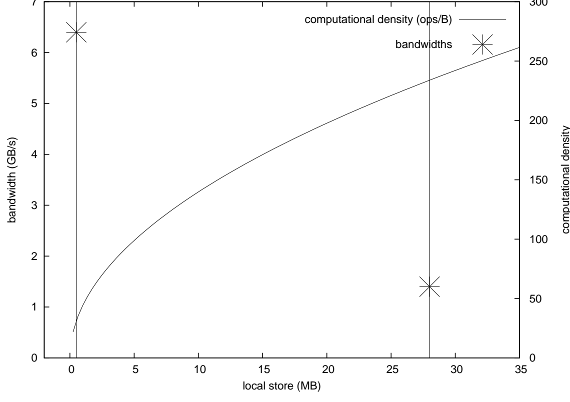

Figure 1: Graph of $\rho_B(\alpha)$ and values of β.

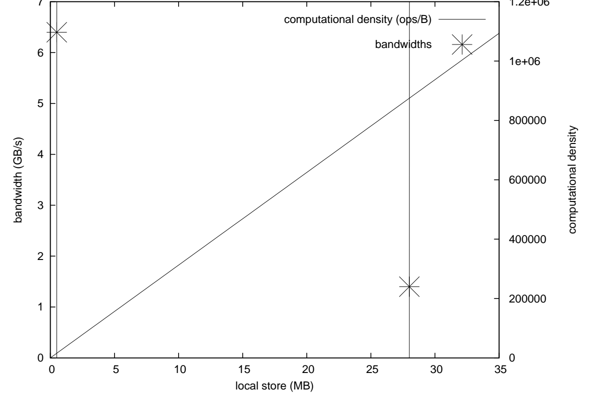

Figure 2: Graph of $\rho_C(\alpha)$ and values of β.

so the calculation speed will depend on both β and μ.

Assuming again that the operands are 4-byte values, for the innermost communication layer, between the OBM and the FPGA,

$$\sigma_{B,1} = \rho_B(\mu_1)\beta_1 = \rho_B(.6\ MB)(6.4\ GB/s)$$

$$\sigma_{B,1} = 219\ G\ ops/sec \tag{9}$$

Calculation speed for the communications between main RAM and the Map:

$$\sigma_{B,2} = \rho_B(\mu_2)\beta_2 = \rho_B(24\ MB)(1.4\ GB/s)$$

$$\sigma_{B,2} = 303\ G\ ops/sec$$

The results of the estimation of algorithm B paint a very different picture than algorithm A. If these were the calculational speeds of floating point operations, the limits would be 220 GF, quite respectable for any processor. The important point of this analysis is that due to the shape of the computational density curve, the bottleneck on this calculation is not the link to main memory, but the link between the OBM and the FPGAs in the MAP processor, although the two are fairly balanced for this problem topology.

Figure 1 illustrates the increasing ρ function and decreasing values of β that contribute to almost uniform values of σ across the different memory interfaces.

4.4 Algorithm C on Map-C

For the all-to-all interaction, assume that each particle takes up 32 bytes, 12 bytes for x,y,z position, 12 bytes for accumulated force values and 8 bytes for index data.

$$\rho_C(\alpha) = \frac{\alpha}{2s^2} = \frac{\alpha}{2048\ B}$$

$$\sigma_{C,1} = \rho_C(\mu_1) \cdot \beta_1 = \rho_C(.6\ MB)(6.4\ GB/s)$$

$$\sigma_{C,1} = 1.88 \frac{T\ ops}{sec}$$

$$\sigma_{C,2} = \rho_C(\mu_2) \cdot \beta_2 = \rho_C(28\ MB)(1.4\ GB/s)$$

$$\sigma_{C,2} = 19.1 \frac{T\ ops}{sec}$$

This is an extremely fast computation rate, to any measure. Clearly, the memory system is not the limiting factor in the topology of this computation. The computational density and β values are illustrated in figure 2. The computational density is linear, so climbs much more strongly from .6 to 28 MB, and so is dominant over the decreasing bandwith.

What would the effect be if the input data were part of a larger set of data structures, say, of size 512 B per data point:

$$\rho_C(\alpha) = \frac{\alpha}{2s^2} = \frac{\alpha}{262,144\ B}$$

$$\sigma_{C,1} = \rho_C(\mu_1) \cdot \beta_1 = \rho_C(.6\ MB)(6.4\ GB/s)$$

$$\sigma_{C,1} = 14.7 \frac{G\ ops}{sec}$$

Here the n^2 data topology fits into this data movement scheme so well that it can tolerate 512 B per data point without bogging down in data transfers.

5 Real performance analysis

The performance analysis in this paper is not real performance prediction; rather it targets the general concern of whether or not an algorithm will fit within the memory subsystem that is designed to feed it. Using the SRC MAP-C as a baseline, the answer for the three explored topologies are "No", "Maybe", and "Yes". But an algorithm fitting within the data flow architecture of a system is not a sufficient condition, merely a necessary one. If an algorithm is shown to fit within the data delivery capabilities of an FPGA

hardware architecture, then the much longer task of porting the algorithm to the FPGA system begins.

The ultimate speed, in many cases, will be limited by the number of functional units that can be instantiated on the surface of the FPGA. As an example, Oak Ridge National Laboratory found[13] that the FPGAs on the MAP-C processor could instantiate 25 multiply-accumulate units per FPGA, or 50 in all. The result from equation 9 divided by the 100 MHz clock on the MAP processor shows that the memory architecture is capable of sustaining at least 2000 MAC operations per second. Compared to only being able to instantiate 50 processors, a that matrix multiply on this hardware is overwhelmingly CPU-bound.

6 Conclusions

This paper sets forth a simple formalism to apply to any application to determine its suitability for an FPGA-based architecture, or any accelerator architecture. By combining the application analysis with performance data from a hardware system, any limits that the memory system of that hardware would impose on the running of that algorithm will be discovered. Based on that information, the programmer or scientist can make an informed decision about whether or not the application could possibly be accelerated by such an architecture.

Furthermore, the three applications explored in this paper are indicator algorithms for memory performance. Applications with memory characteristics like algorithm A will tend to be memory bound, like algorithm C will tend to be CPU bound. and like algorithm B will tend to be balanced. This approach can be used even at the design stage of codes and hardware to predict how the resulting algorithm will fit into the computer system.

A Derivation of computational speed function

A given layer in the memory hierarchy has as local store of size μ. To bring the whole input data set of size M in, then, it must be divided up into a number of chunks, $\gamma = \frac{M}{\mu}$. The time to transfer each chunk is $\tau = \lambda + \frac{\mu}{\beta}$. So under ideal conditions, where each byte must be loaded only once ($\varphi = 1$), the total time T_{ideal} to bring all of the data M into the processor is

$$T_{ideal} = \gamma \cdot \tau = \left(\frac{M}{\mu}\right)\left(\lambda + \frac{\mu}{\beta}\right)$$

$$T_{ideal} = M\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)$$

To calculate T. the total time to load all data including reload penalties, just multiply by φ:

$$T(\beta,\mu,\lambda) = T_{ideal}(\beta,\mu,\lambda)\varphi(\mu) = M\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)\varphi(\mu)$$

Solve equation 2 for φ

$$\varphi(\mu) = \frac{Z}{M}\frac{1}{\rho(\mu)}$$

and substitute into T:

$$T(\beta,\mu,\lambda) = M\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)\left(\frac{Z}{M}\frac{1}{\rho(\mu)}\right)$$

$$T(\beta,\mu,\lambda) = Z\left(\frac{1}{\rho(\mu)}\right)\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)$$

The computational speed (as limited by this memory transfer layer) is the total number of computations divided by the total time to do them:

$$\sigma(\beta,\mu,\lambda) = \frac{Z}{T} = \frac{Z}{\left(Z\left(\frac{1}{\rho(\mu)}\right)\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)\right)}$$

Multiply the resulting expression by $\frac{\beta}{\beta}$ to make the denominator into the form $1 + \epsilon$,

$$\sigma(\beta,\mu,\lambda) = \rho(\mu)\frac{1}{\left(\frac{\lambda}{\mu} + \frac{1}{\beta}\right)}\left(\frac{\beta}{\beta}\right)$$

which derives $\frac{\beta\lambda}{\mu}$ and the final equation for σ:

$$\sigma(\beta,\mu,\lambda) = \rho(\mu)\cdot\beta\cdot\frac{1}{\left(1 + \frac{\beta\lambda}{\mu}\right)}$$

References

[1] A. Aggarwal, B. Alpern, A. Chandra, , and M. Snir. A model for hierarchial memory. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 305–314, 1987.

[2] A.Snavely, N.Wolter, and L.Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE 4th Annual Workshop on Workload Characterization*, 2001.

[3] Darren J. Kerbyson and Philip W. Jones. A Performance Model of the Parallel Ocean Program. *Int. J. High Performance Computing Applications*, 2005.

[4] Kevin J. Barker, Scott Pakin, and Darren J. Kerbyson. A Performance Model of the Krak Hydrodynamics Application. In *International Conference on Parallel Processing (ICPP 2006)*, Columbus, Ohio, 2006.

[5] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kale. Namd: Biomolecular simulation on thousands of processors. In *Supercomputing*, 2002.

[6] Steven Gottlieb. Benchmarking and tuning the milc code on clusters and supercomputers. *Nucl. Phys. B*, 106-107:1031–1033, 2002.

[7] J. Weinberg, M. O. MCracken, a. Snavely, and E. Strohmaierm. Quantifying locality in the memory access patterns of hpc applications. In *Supercomputing*, 2005.

[8] J. Hennessy and D. Patterson. *Computer architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[9] Src Computers Incorporated. http://www.srccomp.com.

[10] *SRC C Programming Environment v2.1 Guide*. SRC Computers Inc.

[11] Jonathan Rose and Dwight Hill. Architectural and physical design challenges for one-million gate fpgas and beyond. In *Proceedings of the 1997 ACM fifth international symposium on Field-programmable gate arrays*, pages 129–132, 1997.

[12] Jon Huppenthal (SRC Computers). personal e-mail. 20 μs latency measured at NCSA, other numbers from SRC. Thank you for letting us use this information.

[13] M. C. Smith, J. S. Vetter, and S. R. Alam. Scientific computing beyond cpus: Fpga implementations of common scientific kernels. In *MAPLD*, 2005.