Comparing the implementation of two-dimensional numerical quadrature on GPU, FPGA and ClearSpeed systems to study electron scattering by atoms

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1733

Comparing the implementation of two-dimensional numerical quadrature on GPU, FPGA and ClearSpeed systems to study electron scattering by atoms

C. J. Gillan[1,*,†], T. Steinke[2], J. Bock[2], S. Borchert[2], I. Spence[1] and N. S. Scott[3]

[1]*The Institute of Electronics, Communications and Information Technology, Queen's University Belfast, The Northern Ireland Science Park, Belfast, BT3 9DT, U.K.*
[2]*Konrad-Zuse-Zentrum für Informationstechnik Berlin, Takustrasse 7, D-14195 Berlin-Dahlem, Germany*
[3]*The School of Electronics, Electrical Engineering and Computer Science, The Queen's University of Belfast University Road, Belfast, BT7 1NN, Northern Ireland, U.K.*

SUMMARY

The use of accelerators, with compute architectures different and distinct from the CPU, has become a new research frontier in high-performance computing over the past five years. This paper is a case study on how the instruction-level parallelism offered by three accelerator technologies, FPGA, GPU and ClearSpeed, can be exploited in atomic physics. The algorithm studied is the evaluation of two electron integrals, using direct numerical quadrature, a task that arises in the study of intermediate energy electron scattering by hydrogen atoms. The results of our 'productivity' study show that while each accelerator is viable, there are considerable differences in the implementation strategies that must be followed on each.

Received 31 August 2010; Accepted 30 January 2011

KEY WORDS: numerical integration; quadrature; accelerator; FPGA; GPU; ClearSpeed

1. INTRODUCTION

The need for heterogeneous chips in high-performance numerical computing was identified by Chillemi *et al.* in 2001 [1]. The application of performance accelerators is not actually a new concept. In the 1980s the Intel 8087 was a co-processor for the x86 CPU line while matrix modules were available on IBM mainframes. Heterogeneous accelerator technologies for numerical computation became widely available as CPU peripherals approximately around 2006. At the software level, these represent a suitable abstraction, notwithstanding the inherent data bus latencies, as a precursor to the fully integrated single chip solutions. The future trend is towards heterogeneous instruction set processors with multi-core CPUs as one component but with custom data paths intermixed. For example, the AMD Fusion program [2] aims to deliver CPU and GPU capabilities integrated into a single-die processor. Reconfigurable hardware also plays a role in such systems. The Intel-Xilinx-Nallatech Slipstream module [3] is one example which is plug compatible with Xeon slots on a motherboard. The viability of reconfigurable computing as a co-processor for a CPU was considered nearly 50 years ago at UCLA [4, 5], but the technology to implement it was not then available [6]. Recently, Convey have reported [7] on substantial performance gains with their hybrid architecture HC-1, which is composed of an Intel Xeon processor and multiple accelerator engines based on Xilinx Field Programmable Gate Arrays (FPGAs).

*Correspondence to: C. J. Gillan, The Institute of Electronics, Communications and Information Technology, Queen's University Belfast, The Northern Ireland Science Park, Belfast, BT3 9DT, U.K.
†E-mail: c.gillan@ecit.qub.ac.uk

We focus on three accelerator technologies in this paper: the Virtex-4 FPGA-based RC-100 product from Silicon Graphics, several of the CUDA-enabled NVidia GPU products and the Advance 620e accelerator card from ClearSpeed. All of these architectures offer instruction-level parallelism and this has to be exploited to realize performance gains with respect to any CPU implementation.

From the perspective of an application scientist the key metric against which any new compute technology will be judged is the ease with which it can be programmed (productivity). The ideal would be to take existing legacy code, mostly written in a high-level language such as Fortran and most likely developed over generations, and compile it unmodified to run on the combined CPU-accelerator environment. Put simply, this is not yet possible so that the programming languages, compiler tools and system components for accelerators represent a significant barrier to entry for many. One reason for this is that creating an automatically parallelizing compiler is a staggeringly difficult task, as seen by the intractability of the problem in the face of the huge resources that have been targeted at it [8, 9]. The European Network of Excellence HiPEAC have listed in some detail the challenges facing the field [10].

By the time an algorithm is expressed in, for example Fortran or C, it involves programming features such as indirect addressing, pointers, recursion, indirect function calls and possibly accesses to global resources. It then becomes essentially impossible for an automated tool to unravel the true nature of the algorithm. It follows that when a piece of legacy code is to be implemented to run on a heterogeneous accelerator platform, it is generally necessary for a human to perform the process of reverse engineering the algorithm and reformulating it for the new environment. We have sought to evaluate this process in this paper in respect of one algorithm. We chose to look at a computational hotspot in 2DRMP [11] which is a variant of the R-matrix method [12, 13], a HPC application that has been developed for over 30 years in the Fortran language.

The paper begins by defining the integrals to be evaluated. These are an essential initial step in the computational modeling of the electron scattering process. Subsequent steps in the computational model use these integrals to construct, for example, matrices representing the interaction between incident particle and the target atom. The section following that briefly presents the programming environment which is used on each accelerator system. Ultimately, this paper is about the application of those environments to derive performance from an initial Fortran source. The common theme is the porting of this Fortran source, but in different ways each time, to each of the three distinct accelerator platforms and programming environments. This means that we treat the accelerators independently in this paper; there is currently no one-size-fits-all solution to our porting problem. For example, the FPGA system offers the flexibility of selecting arbitrary bitwidths for floating point representations, whereas the ClearSpeed card is IEEE single and double precision and the GPU, at least for one of the cards used here, offers only single precision floating point operations in hardware. Details of the accelerator hardware for each system are presented later in the paper. We have included a section on the use of Posix threads on the CPU for the purpose of completeness. Subsequent sections of the paper report on the implementation of the algorithm on each accelerator and present execution time results. The paper closes with come conclusions and looks forward to the new generations of accelerator hardware.

Our work on integration is similar in nature to the evaluation of two electron integrals for molecular scattering [13] and structure computations [14], a topic that has been addressed on the both GPU [15] and FPGA [16] and ClearSpeed systems [17]. The single center expansion technique, an alternative to the R-matrix approach to solving the scattering problem for molecular targets, has also been recently upgraded to use GPUs [18].

2. THEORETICAL BACKGROUND

The background to R-matrix theory [12, 13] is not presented here, but may be found in the work by other authors [11] and in the references therein. A rate-limiting step in 2DRMP is found to be

DOI: 10.1002/cpe

the evaluation of two electron integrals of the form

$$I(n_1,l_1,n_2,l_2,n_3,l_3,n_4,l_4,\lambda)=\int_0^a u_{n_1l_1}(r)u_{n_3l_3}(r)\left[\frac{I_1}{r^{\lambda+1}}+r^{\lambda}I_2\right]\mathrm{d}r \tag{1}$$

where the inner integrals, I_1 and I_2, are defined by

$$I_1=\int_0^r t^{\lambda}u_{n_2l_2}(t)u_{n_4l_4}(t)\,\mathrm{d}t \quad \text{and} \quad I_2=\int_a^r \frac{1}{t^{\lambda+1}}u_{n_2l_2}(t)u_{n_4l_4}(t)\,\mathrm{d}t \tag{2}$$

The list of tuples $(n_1,l_1,n_2,l_2,n_3,l_3,n_4,l_4,\lambda)$ depend on the overall symmetry quantum numbers, both spatial and spin, of the system but can be computed in seconds using integer arithmetic. The functions u_{nl} are precomputed on a fixed spaced grid in $[0,a]$ by solving a classical Sturm–Liouville type eigenvalue equation, which represents a one-electron model scattering problem. These functions are oscillatory, reflecting the continuum wave nature of the scattering problem. Performing the large number of numerical integrals required by the physics of the problem is a challenging numerical and computational task in part due to the oscillatory nature of the functions. Computation of the integrals for one scattering symmetry may require many hours computing on a single CPU core. The R-matrix boundary, a, is typically taken to be 15 Bohr radii. Analysis using a dynamical error estimation method [19] implemented in the CADNA toolset [20] has shown that in this case the grid must have 1191 points in order to generate sufficient numerical accuracy [11].

From a data-driven perspective, the computational task defined by Equation (1) consists of processing data from two lists to produce a further list:

1. The set of functions u can be arranged in memory as a list with each member composed of 1191 floating point numbers. For a typical case, there are 160 functions thereby requiring 0.73 MB of storage if held in the IEEE 754 single precision format.
2. The list of all tuples can be generated and stored in memory and then each member of this list is processed independently. The tuple can be stored as a set of eight bit integers. In essence each tuple contains four pointers into the list of functions u because each n,l can be compressed into a single index. In a calculation where the tuple list has 10^8 entries then 0.5 GB of storage is needed to hold it.
3. The computation for each tuple consists of performing sum reductions that eventually produce just one value. Thus, the end product of the complete calculation is a further list, with one value per integral.

The I_1 and I_2 are computed for every point in the mesh using the Simpson three-point rule, that is using a three-point stencil as we move along the mesh. This is an inherently sequential process as we require to add the previous integrals as we move through the mesh. Also, it is more stable to compute I_2 by integration from the boundary a inwards to r [11]. This has a significant impact on the FPGA design as explained below. Once the vectors I_1 and I_2 are computed, the outer integral is evaluated using an 1191 point Simpson rule; this final step is a sum reduction over all points of the computed integrand, where the computation at each point requires several multiplication operations.

3. ACCELERATOR PROGRAMMING ENVIRONMENTS USED IN THE COMPUTATIONS

The R-matrix program that we started with was written in Fortran. This is the programming language that is used to benchmark the fastest supercomputers and with over 50 years of development having taken place in the language, we recognize that the task of porting Fortran to new accelerator technology will be faced by many programmers.

All accelerators at present operate symbiotically with a CPU-based host. This means that the key to using accelerators for performance gains is to identify hotspots in the CPU code and to move those hotspots into kernels running on the accelerators, but only those kernels that are suitable for execution on the accelerator. A code that involves nested and complicated if-then-else blocks, that

DOI: 10.1002/cpe

is to say a code dominated by logical operations and therefore branching, is not well suited for execution on the current generation of accelerators.

When profiled with *gprof* on Linux, the routine that performs the integrals discussed earlier in this paper accounted for over 90% of the execution time. We therefore rewrote the integration routine in ANSI C. This was done manually because it was found to be the best way to generate an efficient code. We then tested the ANSI C program on various CPU systems to ensure that the results generated were indeed the same as those obtained from the Fortran version. Thereafter, the C procedure which computed the integrals, became the base code which was migrated to each accelerator architecture.

Dialects of the C programming language are available for all three accelerators. Each is unique and employs new concepts as explained in the following list:

1. *CUDA* is a function-pointer-free and recursion-free subset of the C language with additional syntax expressions and attributes for expressing the memory hierarchy. It is developed by NVidia for use with their GPUs.
2. C^n is a data-parallel extension to the C language created by ClearSpeed for programming their equipment. Parallelism is most often expressed by using the qualifier *poly* when declaring the data type and not by specific code instructions.
3. *Mitrion-C* is a data-parallel language, employing the C syntax, but with entirely different semantics to ANSI-C. It is available on Cray, Nallatech and Silicon Graphics FPGA-based systems. Mitrion-C code is transformed into a custom processor, the Mitrion Virtual Processor, matching the algorithm as specified in the code.

While an expert C programmer can become active with these new programming environments within a day, we found that there is a significant learning curve, in our experience of several weeks duration, to be traversed in order to be able to write highly optimized code with any of the language extensions.

We considered the possibility of using Fortran to CUDA translators such as f2c-acc [21] and the CUDA Fortran extension to the Portland Group compiler. However, insofar as we are aware no similar translators exist for FPGA or ClearSpeed systems, so we do not report on this approach.

For all three accelerators, the technique that we adopted was to package up the lists of information to be processed and to copy these to the accelerator device in batches then launching the computations. This model is illustrated schematically in Figure 1.

The ClearSpeed Advance accelerators are designed specifically as high-performance engines for floating point numerical work (at the time of their market introduction) and were shipped with a customized implementation of several linear algebra routines, in particular DGEMM. We attempted to exploit this by recasting the CPU code in terms of DGEMM. The use of the ClearSpeed card then becomes just a matter of linking against the ClearSpeed-Math-Library CSXL. A side effect of this is that we can do exactly the same for CUBLAS, NVidia's version of BLAS albeit that this is only available in single precision for the GeForce 8800.

3.1. GPU systems

We used two different types of NVidia GPU systems. Early work was performed on a quad-core Dell Workstation, which had two NVidia GeForce 8800 GTX GPUs located on the PCI Express bus. The 8800 GTX has 16 multi-processors and each multi-processor in turn is a cluster of eight processing elements (PEs), giving a total of 128 processors on the card. Only single precision floating point arithmetic is available. There is a memory hierarchy on the GPU cards:

- Each PE also has its own local memory. In fact these are 32 bit registers that are used in the arithmetic operations. A 32 kB register file is evenly distributed across all active threads within a multi-processor.
- Each multi-processor (8 PEs) has its own 16 kB shared memory that can be accessed by all its PEs.

DOI: 10.1002/cpe

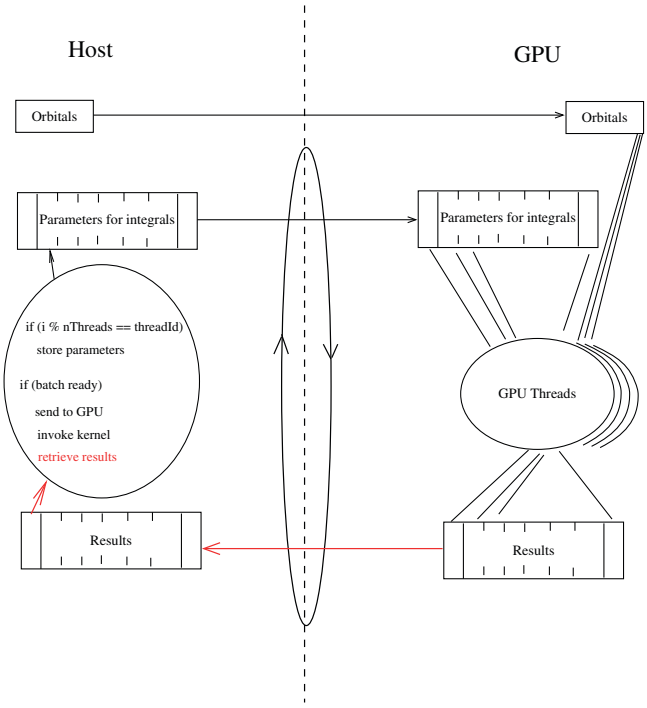

Figure 1. A schematic illustration of the mechanism for managing the computation process on the GPU accelerators. In the FPGA case, the GPU threads are replaced by a network of processing elements.

- There is a 768 MB memory available to all PEs in all multi-processors; that is an on-card global memory. The bandwidth to this memory is specified as $86\,\mathrm{GB\,s^{-1}}$, but it carries a penalty of a 400–600 clock cycle delay.

Later work employed an NVidia S1070 system attached to a Dell cluster. The S1070 has 4 Tesla GPUs in a 1U rack mount each with 240 processor cores. A distinction from the GTX 8800 processors is that a double precision arithmetic unit is included in each multi-processor. The S1070 is also attached to the cluster by PCI Express x16 cables which allow an aggregate data transfer rate of $8\,\mathrm{GB\,s^{-1}}$ bi-directionally. The memory hierarchy is identical but different in size. For example each of the GPUs has 4 GB DRAM for its global memory.

3.2. FPGA system

Field programmable gate arrays (FPGAs) are composed of fixed numbers of configurable logic blocks (CLBs) in a spatial mesh, attached to which are also higher logic such as 18×18 bit multipliers and block RAMs. The logic blocks can be configured to implement boolean functions and these can be connected with each other and with the other components in a network to create a customized processor. In essence, a network of arithmetical–logical units is created with data flowing from element to element under the control of clock signal.

Our FPGA platform was an SGI Altix system with two RC-100 blades. The RC-100 blades are attached to the host through the proprietary SGI NUMAlink interconnect. Each RC-100 contains two user programmable Xilinx Virtex-4 LX200 FPGAs. The LX200 has 200 448 CLBs, 96 18×18 bit mulipliers and 6048 kbits of ECC block RAM. Each LX200 on the RC-100 blade has an attached 40 MB SRAM arranged as 128 bit cells and this is analogous to the global memory on each GPU card. The SRAM is connected to the Altix fabric by the proprietary SGI NUMAlink 4 protocol offering a single link aggregate transfer rate of $3.2\,\mathrm{GB\,s^{-1}}$ in each direction.

We expressed our algorithm in Mitrion-C, which is a compiler technology, designed for the HPC market. The semantics of Mitrion-C reflect the both the spatial and temporal nature of the FPGA.

DOI: 10.1002/cpe

The Mitrion compiler works with the Xilinx ISE toolset to create a bitstream, at a fixed clock rate of 100 MHz, which is then loaded onto one or both FPGAs on the RC-100 for execution. In this work we use Mitrion-C release 1.5.1. Our strategy is similar to the work of other authors with the RC-100 [22].

We used the two FPGAs on the blade in a similar fashion to the four GPUs within the S1070 dedicating a CPU thread to handle each. On the host side, we have used the MITHAL library from Mitrionics to control the FPGA and the transfer of data via NUMALink.

3.3. ClearSpeed system

The ClearSpeed Advance is a 620e card is connected via PCI Express (x8) and is equipped with two CSX600 processor. Each CSX600 contains a mono execution unit (Mono-Unit) and 96 processing cores that are aggregated into a multi-threaded array processor (MTAP). Each of the 96 cores is referred to as a PE. Each MTAP contains 128 kB ESRAM, which can be set up to hold code and data. The *poly* extension to the data type declaration in the C^n language, as mentioned above, is used by the compiler to map execution onto the PEs. Scalar data type are processed in the Mono-Unit. The clock speed of the MTAP on the CSX600 is 210 MHz. This low clock speed correlates with the reported peak power consumption of 33 W for the card.

The PEs each contain a 32/64-bit floating point multiplier unit and adder unit and each has 6 kB of local SRAM. There are 512 MB of RAM per MTAP (or 1 GB in total) on the ClearSpeed e620 card serving as a global memory for all its PEs and this memory is ECC protected. The 620e, which we used, was installed in a Sun 4600 M2 workstation with eight dual-core Opteron 8220 CPUs.

4. POSIX THREADS IMPLEMENTATION ON THE CPU

The integrals computation that is addressed in this paper is embarrassingly parallel and therefore likely to be a good candidate for the application of thread parallelism. This means the use of the Single Program Multiple Data paradigm (SPMD) or more specifically the Work Crew paradigm as defined by Butenhof [23]. We implemented this using the Posix thread library functions and not OpenMP. This was because we wanted to have a fine-grain control for thread management. In the case of the integrals, individual threads are not synchronized at the instruction level and this permits a simple load balancing strategy to be followed in order to distribute integrals among the threads. In pseudo-code this is:

```
for(itup=1,num_tuples)
    if( modulus(itup,max_threads) == this_thread_index)
       evaluate integral for itup
```

We executed the code for several different lengths of the tuple list (i.e. differing numbers of integrals) and used the timing data for statistical analysis. The mean time to compute one integral in a single threaded implementation was 1.83×10^{-5} s. In the case of multiple threads (>8) on 16 CPU system, we found that the elapsed execution time can be fitted to the form

$$T = 1.16\times10^{-6}\times\textbf{Number of integrals} \tag{3}$$

This linear scaling reflects, in our opinion, the embarrassingly parallel nature of the problem. The expression for T does not extend towards smaller numbers of threads and this is why the mean time per integral in a single threaded implementation, that is a sequential implementation, is much larger. The housekeeping work needed to set up the data for the integral computations is being amortized over the larger numbers of threads.

5. DGEMM FORMULATION

We investigated an alternative approach in which we reordered the list of tuples into blocks labeled by the λ parameter [24]. With respect to each batch, the radial powers are now fixed qualities. Each batch of outer integrals, Equation (1), can then be expressed as a matrix multiply, **UV** where

DOI: 10.1002/cpe

the matrix $\mathbf{U}$ is of dimension $n \times 1191$ and the matrix $\mathbf{V}$ is of dimension $1191 \times m$. n is the number of distinct $u_1 u_3$ products in the batch and correspondingly m for $u_2 u_4$.

The matrix product can be computed by a library call to DGEMM. The ClearSpeed accelerator ships with a customized DGEMM implementation which also performs work on the CPU concurrently with the CSX600. We linked an optimized version of the Atlas library into the executable to handle the CPU side of this DGEMM work. The load split factor is tunable and we optimized it on our system so that about 10% of the DGEMM load was CPU side. Essentially the same CPU code could be recompiled and linked against the NVidia CUBLAS implementation for the GeForce 8800s, in this case using SGEMM. Insofar as we are aware there is no CPU computing concurrently with the GPU in CUBLAS.

While this strategy required minimal understanding of the 620e system, and no programming in C^n, unfortunately we found that the performance of the overall program was reduced relatively to the single threaded version. This is due to two features:

- There is a substantial amount of data movement activity on the CPU to prepare the $\mathbf{U}$ and $\mathbf{V}$ in the format needed leading to wasted CPU cycles.
- The n, m values, even for higher λ, are too small to leverage the performance of the optimized DGEMM, SGEMM implementations. Moving to smaller grid point spacing or extending the upper limit for λ may serve to improve this situation, although it would mitigate the previous point too.

6. IMPLEMENTATION ON NVIDIA GPU SYSTEMS

The threading model used on NVidia GPUs is significantly different from that available with Posix on the CPU: the distinction lying in the hardware. One key distinction is that all threads in a thread block being executed on a multi-processor operate in the SIMD fashion and another key point is the number of threads launched is significantly larger than on a CPU. Consideration of the architecture of NVidia GPUs leads to several observations for the implementation of our algorithm, as follows:

1. We cannot store the I_1 and I_2 results, for each integral, in vectors. Doing so, when summed over all threads in a block, creates an unsustainable register pressure on each multi-processor and indeed is also not viable in the shared memory available. The amount of on-chip memory available to each thread within the block is therefore a limiting factor. Simply storing these in global memory, assuming enough is available, is not a solution either because of the atency of access.
2. The Single Instruction Multiple Data model of parallelism means we cannot use the if statement approach to distinguish threads as we have done on the CPU because it is not an optimum construction in that environment.
3. The u_{nl} functions are used repeatedly in different integrals but are read-only data. Additionally, the functions are accessed element by element in the integral computation. This permits the use of texture memory, as a specific feature of global memory. Texture memory, in the context of numerical computation, essentially offers an optimized path to a region of global memory which is being used as read-only. A substantial performance benefit can be gained from texture memory by having all the threads in a warp access nearby locations in the texture.
4. For the GTX 8800, only IEEE 754 single precision floating point arithmetic is available. On the Tesla system there is one IEEE 754 double precision unit on each multi-processor, whereas there are eight single precision units. The radial powers and inverse powers in the integrals require careful handling in single precision as they generate over and under flow.
5. On the S1070 Tesla system, we need to employ all four GPUs concurrently from the CPU, if we are to derive maximum performance. The best way to do this is to employ Posix threads on the CPU, each independently controlling one GPU within the S1070 box. In addition, we have executed the code in a similar way on the system with two GTX 8800 cards.

DOI: 10.1002/cpe

The consequence of the above points is that we had to substantially rewrite the procedure that performs the integration to gain performance on the GPUs processing elements.

6.1. Results

Modification and testing of the code required a few weeks effort once a basic understanding of the GPU hardware and the CUDA language was acquired. In IEEE754 single precision execution of the code equated to a mean time per integral evaluation of 1.25×10^{-7} s on the 2x GTX 8800 system. The numerical accuracy of the results was in error by 0.0001% with respect to the computed IEEE754 double precision values corresponding to an absolute error of 1×10^{-7}.

Executing the code in double precision on the S1070, that is with four Tesla GPUs, we found that the speed is about half the single precision speed of using two GTX 8800s. This reflects the fact that Tesla GPUs have 16 double precision units, whereas two GTX 8800s have 256 single precision units. An additional factor which explains this difference in timings is that the clock speed of the Tesla GPUs in the S1070 is reported to be double that of the GTX 8800 GPUs.

7. IMPLEMENTATION ON AN FPGA

Equation (1) defines the arithmetical units that are required for the computation so the main challenge of implementing this on the FPGA is designing the network of elements using the syntax of Mitrion-C. The two mechanisms that we can use to leverage instruction-level parallelism are pipelines and wide parallelism. Each involves consuming space on the FPGA and each has different latency associated with it.

The design is bounded not only by the resources consumed but also by the capacity of the Xilinx tools to connect these together within the constraint of a 100-MHz clock signal. By exploiting the binary representation format of floating point numbers, the multiplication by 2 and by 4 that arise in Simpson's rule are implemented in minimal resources by using only bit masks and shifts.

In Mitrion-C we read the data from global SRAM subject to the limitation that we can read only one SRAM location per clock cycle. The data is then fed through the network of compute elements. Figure 2 is a schematic illustration of the implementation. In the left-hand model (F1), I_1 and I_2 are computed in parallel and their contributions to (1) are also computed in parallel. This requires arithmetical-logical units for the Simpson rule to be created for both cases. For the right-hand model (F2), we create just one instance of the arithmetical-logical units, but twice as much data has to pass through each and this can be processed in a pipelined fashion.

The FPGA gives the user the ability to control the number of bits used for floating point numbers. We are no longer restricted to only IEEE 754 single and double precision data formats. In our code all data read from (and written to) the SRAM banks is converted to (from) the type used in

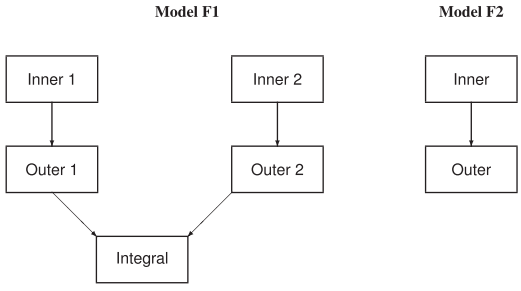

Figure 2. This illustrates schematically the arrangement of a network for computation of the integrals in terms of pipelining and wide parallelism on FPGA. The design on the left-hand side uses five blocks to deliver two parallel pipelines. The right-hand side option is less parallel but has a longer pipeline structure. As reflected by the picture, the consumption of spatial resources on the device is significantly different.

DOI: 10.1002/cpe

Table I. The resources consumed on the Virtex-4 LX200 and mean compute time per integral for the two models presented in Figure 2.

Model	Precision	Resources used on the FPGA			Execution time (s)
		Flip-flops (%)	BlockRAM (%)	Multipliers (%)	
F1	Double	61	61	69	N/A
F1	Half-double	32	61	65	4.45×10^{-5}
F2	Double	45	52	69	12×10^{-5}

F1 was investigated for two bitwidths, but the Xilinx tools could not synthesize a bitstream for IEEE 754 double precision and hence no execution time is reported. The time per integral incorporates the concurrent use of the two FPGAs on the RC-100 board.

our code for computation in order to be compatible with the fixed sizes offered by the CPU. This variability further influences the spatial resource consumption aspect.

7.1. Results

We experimented with many combinations of bitwidth selections for each of the models F1 and F2 discussed above. In each case we checked the numerical accuracy relative to the results on the CPU. We found that IEEE754 half-double precision was sufficient for eight significant figures of accuracy. This is adequate for the scattering problem. This uses 43 bits total with 11 dedicated to the exponent. Although defined as a standard, it has never been implemented in a CPU as far as we are aware.

The execution times are lower than obtained on the GPU system. It is not surprising that model F1, which has wide parallelism therefore uses more FPGA resource and executes more quickly. The smaller the bitwidth, the fewer the number of barrel shifters required in a floating point unit and this leading lower latency in the circuit. This effect contributes to the timing reported in Table I for model F1.

It is well known that FPGAs are well suited to fixed point computations [25]. In the integrals problem studied here, the orbitals u_{nl} are computed to be orthonormal and are normalized to unity and this is ideal for a fixed point implementation. The presence of radial powers and inverse powers in Equations (2), (1) complicates the formulation of the current problem in fixed point format. As we have indicated, they cause arithmetic overflow and underflow in single precision floating point arithmetic. We have derived a 40 bit fixed point version of the algorithm but have not as yet implemented it.

8. IMPLEMENTATION ON CLEARSPEED

The similarity of the memory hierarchy between ClearSpeed and the GPU systems means that similar strategies can be followed to gain performance from the e620 card. One difference is that the MTAP processor has only two levels of memory whereas the GPU has several.

The numerical orbitals and list of tuples are initially stored in global memory after being downloaded from the host system. Each PE performs the integration for one member of the list of tuples. Because there is not enough memory at the PEs, we are processing 48 grid points at a time, repeating the process until all points in the integration grid have been handled.

We experimented with three models differing in how we loaded the data onto the PEs. In model C1 three of the orbitals in each integral were moved from ESRAM of the Mono-Unit into SRAM for each PE as required. Then in model C2, all four were moved from ESRAM to SRAM of each PE, while in model C3 the orbitals were simply left in the global memory of the attached to the MTAP. We should point out that no vector extensions were used in our C^n code and this is clearly an area for future study.

DOI: 10.1002/cpe

Table II. Mean compute time per integral on ClearSpeed system.

Model	Mean time per integral (s)
C1	1.32×10^{-5}
C2	1.76×10^{-5}
C3	2.38×10^{-5}

8.1. Results

Table II reports the mean time to compute one integral for each of the three models defined in the previous section. The ClearSpeed software development environment includes a comprehensive profiler tool and we were able to use this to visualize the performance of the system. We noted the following behavior during execution of the integrals code written in C^n:

- If we have four different indexes (C2 orbitals in ESRAM and C3 orbitals in DRAM) most of the time we are not calculating integrals, but waiting for copy operations to finish. This shows that memory layout is very important. However there is not enough ESRAM on the MTAP to hold the whole orbital array, so when possible one should prefetch required data from DRAM to ESRAM (and then copy to SRAM of the PEs).
- If we have only three different indexes (C1), then all 96 PEs use another static index, so that the fourth index is fetched from ESRAM of the Mono-Unit of the Clearspeed MTAP and not from the PE-SRAM, so we have one copy operation less.

Our analysis led to the conclusion that it would be better to presort the tuples, an approach that is similar in concept but different in implementation to the one that we had tried for the DGEMM implementation on the CPU. Using an appropriate sorting algorithm, the calculation of the integrals can then be performed using the most efficient memory access pattern for the orbitals.

9. CONCLUDING REMARKS

With over 50 years development in the language and with its widespread use in high-performance numerical computing and therefore in supercomputing, a great deal of software has been written in the Fortran language. For the new field of accelerator technology this presents a problem in the sense that C dialects are the preferred implementation language for accelerators. This may be a transient problem as the Fortran language evolves but it exists at the time of writing this paper. So, we have studied the translation of one hotspot in a Fortran program into three different kernels for use on three different accelerators.

We started by translating the Fortran program into ANSI-C thereby enabling direct use of Posix threads for the single computational hotspot, an integration routine, in the implementation. We explored the possibility of rewriting the algorithm to use linear algebra libraries for the accelerators but found this to be unsuitable. We have hand coded the floating point integration kernel onto an FPGA using the Mitrion-C language, onto a pair of NVidia GPUs in CUDA C and also in C^n for the ClearSpeed chip. We found that we needed to rewrite the mathematical equations to achieve adequate performance in each case and needed to pay particular attention to the memory hierarchy in the GPU and ClearSpeed cases to achieve optimal performance.

The S1070 proved to be the fastest accelerator of the three for this problem. Numerical integration is inherently a sum reduction activity and this translates to a low ratio of compute to load/store operations. The very large number of threads used in the CUDA implementation is capable of masking the latency of reading the orbitals from texture memory. The execution times for each integral on the FPGA and ClearSpeed were slower than the CPU; however, we suggest that these are still viable co-processors with the CPU. The flexibility of using different word lengths on the

DOI: 10.1002/cpe

FPGA in any case forces one to think again about the numerical characteristics really needed for any implementation. It is also worth remembering the next generation of FPGAs, such as the Altera Stratix-III, will have substantially more components than the Virtex 4 components on the RC-100 that we have used in this work. For example the Altera Stratix-III E260, which is a key component in the Novo-G machine being commissioned at CHREC [26], has 768 18x18 multipliers compared to the 96 available in the Virtex-4 LX200 used in this work. Put simply, these extra resources mean that multiple integrals could be computed simultaneously.

A distinct difference between the three technologies studied is the reported power consumption metrics. However, this relates to the clock speed of the processors in each case. The compute performance of the RC-100, at 100 MHz and the e620 at 210 MHz is similar. The RC-100 blade is reported by the manufacturers to consume about 150 W and each MTAP on the ClearSpeed e620 about 10 W peak per MTAP. Both are significantly lower than the GPU systems where the shader clocks run at speeds in the region of 1.3 GHz. The manufacturer's specification for the S1070 indicates a maximum typical power consumption figure of around 800 W [27]. The overall relationship between power and performance has been studied with Tesla systems for various applications [28]. It should be remembered that the power consumption of any accelerator is additional to the host computer to which it is attached.

Our algorithm suited the NVidia GPU architecture very well because we had multiple independent threads sharing only memory read operations. More generally, GPU programming requires careful consideration of whether the algorithm can be implemented in an SIMD environment.

There are clearly a number of ways in which the research we have reported can be extended. It is only limitations of time that has prevented us from following some of the strategies mentioned above in more detail. There are several high-level language (C to gates) technologies available other than Mitrion-C . We chose the latter however because it is customized for the HPC applications. Even within this choice one could study the impact of using other synthesis tools. In any case, the HPC market continues to evolve and with the imminent availability of newer processor chips, such as the Tilera Gx and Fujitsu Venus chip, we plan to revisit this work in the near future.

ACKNOWLEDGEMENTS

We thank Matthias-Foquet Lapar from SGI and Stefan Möhl from Mitrionics AB for their help with various aspects of the hardware and toolsets for the RC-100. C. J. G. is grateful to the Konrad-Zuse-Zentrum für Informationstechnik Berlin for hosting a visit by him and permitting extended use of the RC-100 and ClearSpeed systems there. Funding for this visit under EPSRC grant EP/G00210X/1 is acknowledged.

REFERENCES

1. Chillemi G, Rosati M, Sanna N. The role of computer technology in applied computational chemical-physics. *Computer Physics Communications* 2001; **139**(1):1.
2. Brookwood N. AMD Fusion: Family of APUs: Enabling a Superior, Immersive PC Experience. An AMD Whitepaper. Available at: http://sites.amd.com/us/Documents/48423B_fusion_whitepaper_WEB.pdf [2 September 2010].
3. Available at: http://www.hpcwire.com/offthewire/17902759.html [9 September 2010].
4. Estrin G. Organization of computer systems: The fixed plus variable structure computer. *Proceedings of the Western Joint Computer Conference*, *Western Joint Computer Conference*, New York, 1960; 33–40.
5. Estrin G. Reconfigurable computer origins: The UCLA fixed-plus-variable (F+V) structure computer. *IEEE Annals of the History of Computing* 2002; **24**(4):3–9. DOI: http://dx.doi.org/10.1109/MAHC.2002.1114865.
6. Lysaght P, Subrahmanyam PA. Guest Editors' introduction: Advances in configurable computing. *IEEE Design and Test of Computers* 2005; **22**(2):85–89.
7. Paulson LD. Start-up unveils flexible supercomputing approach. *Computer* 2009; **42**(1):21–24. ISSN: 0018-9162.
8. Wolfe M. *High-performance Compilers for Parallel Computers*. Addison-Wesley: Reading, MA, 1996.
9. Bik A, Gannon D. Automatically exploiting implicit parallelism in Java. *Concurrency*: *Practice and Experience* 1997; **9**:579–619.
10. De Bosschere K, Luk W, Martorelli X, Navarro N, O'Boyle M, Pnevmatikatos D, Ramirez A, Sainrat P, Seznec A, Stenström P, Tenam O. *High-performance Embedded Architecture and Compilation Roadmap* (*Lecture Notes in Computer Science*, vol. 4050). Springer: Berlin, 2007. DOI: 10.1007/978-3-540-71528-3.
11. Scott NS, Scott MP, Burke P, Stitt T, Faro-Maza V, Denis C. 2DRMP: A suite of two-dimensional R-matrix propagation codes. *Computer Physics Communications* 2009; **180**:2424–2449.

12. Burke PG, Berrington KA. *Atomic and Molecular Processes*. IOP Publishing: Bristol, 1993.
13. Gillan CJ, Tennyson J, Burke PG. *Computational Methods for Electron-Molecule Collisions*, Huo WM, Gianturco FA (eds.). Plenum: New York, 1995; 239.
14. Dupuis M, Rys J, King HF. Evaluation of molecular integrals over Gaussian basis functions. *Journal of Chemical Physics* 1976; **65**:111.
15. Yasuda K. Two-electron integral evaluation on the graphics processor unit. *Journal of Computational Chemistry* 2008; **29**:334–342.
16. Kindratenko V, Ufimtsev I, Martïnez T. Evaluation of two-electron repulsion integrals over Gaussian basis functions on SRC-6 reconfigurable computer. *Proceedings of the Reconfigurable Systems Summer Institute*, 2008. Available at: http://www.rssi2008.org/proceedings/papers/posters/13_Kindratenko.pdf [25 June 2009].
17. Brown P, Woods C, McIntosh-Smith S, Manby FR. Massively multicore parallelization of Kohn–Sham theory. *Journal of Chemical Theory and Computation* 2008; **4**:1620–1626. DOI: 10.1021/ct800261j.
18. Sanna N, Baccarelli I, Morelli G. SCELib3.0: The new revision of SCELib, the parallel computational library of molecular properties in the Single Center Approach. *Computer Physics Communications* 2009; **180**:2544.
19. Vignes J, La Porte M. Error analysis in computing *Information Processing 74*. North-Holland: Amsterdam, 1974.
20. Scott NS, Jézéquel , Denis C, Chesneaux J-M. Numerical 'health check' for scientific codes: The CADNA approach. *Computer Physics Communications* 2007; **176**:507–521.
21. Govett M, Middlecoff J, Henderson T. Running the NIM next-generation weather model on GPUs. *Proceedings of the 10th IEEE/ACM International Conference on Cluster*, *Cloud and Grid Computing*, Parashar M, Buyya R (eds.), Melbourne, Australia, 17–20 May 2010; 792–796.
22. Pocek KL, Buell DL (eds.). *IEEE Symposium on Field-Programmable Custom Computing Machines*, *FCCM 2007*, 23–25 April, Napa, CA, U.S.A. IEEE Computer Society, 2007. ISBN: 0-7695-2940-2.
23. Butenhof D. *Programming with POSIX Threads*. Addison-Wesley: MA, 1997. ISBN: 0-201-63392-2.
24. McLean AD, Yoshimine M. Computation of molecular properties and structure. *IBM Journal of Research and Development* 1968; **12**:206–233.
25. Woods R, McAllister J, Yi Y, Lightbody G. *FPGA-based Implementation of Signal Processing Systems*. Wiley: U.K., 2008. ISBN: 978-0-470-03009-7.
26. Available at: http://www.chrec.org/facilities [25 January 2010].
27. Available at: http://www.nvidia.com/object/product_tesla_s1070_us.html [21 October 2010].
28. Kozin IN. Energy Efficiency Investigation (Power & Performance). *PRACE Workshop New Languages & Future Technology Prototypes*, 1–2 March 2010. Available at: http://www.prace-project.eu/documents/19a_energyefficiency_ik.pdf [22 October 2010].

DOI: 10.1002/cpe