2 Executive summary

The main aim of this project was to improve the performance of the Bristol University Docking Engine (BUDE) program by porting it to run on modern GPUs. BUDE is a drug docking algorithm that predicts the binding energy of two molecules in order to identify potential drug candidates.

The porting was achieved by converting part of the existing code, currently in FORTRAN and C++, to the new OpenCL parallel computing language. The motivation for porting the program was the potential increase in speed, energy efficiency and cost effectiveness that could be provided by moving BUDE onto modern parallel hardware. BUDE had already been ported to use high speed hardware acceleration and was running on ClearSpeed CSX600 processors capable of up to 50 double-precision GFLOPS (GFLOPS = Giga Floating Point Operations Per Second); this accelerator was approximately 10 times faster than a dual core CPU of its era. However the latest GPU architectures such as Nvidia's Fermi are capable of up to 1288 single-precision GFLOPS which this project aimed to exploit.

The secondary aim of the project was to test the program on multiple hardware platforms, including ATI and Nvidia OpenCL-capable GPUs and modern multicore x86-64 processors from Intel and AMD. These tests were performed to find out which platform offers the best absolute performance, performance per watt, performance per dollar, kilowatt hours used per experiment run and C02 emission per experiment run. This not only demonstrates the most cost and power efficient hardware platform for BUDE but also for other real world OpenCL-based programs.

The part of BUDE that took 99.2% of compute time in the original version was the actual energy calculations. This was the part that was successfully ported to OpenCL in this project.

Benchmarks showed a performance increase of 9x when running with the code developed for this project on an Nvidia C2050 'Fermi' compared to the old code running on a ClearSpeed CXS600. The Nvidia GTX280 GPU was identified as the most efficient processor in terms of performance per watt, performance per initial price, kWh per experiment and C02 emissions per experiment.

4 Table of Contents

5 Aims and objectives

The main aim of the project is to improve the performance of the Bristol University Docking Engine (BUDE) program by porting it to run on modern GPUs. This was done by porting part of the existing code, currently in FORTRAN and C++, to the new OpenCL parallel computing language. The motivation for porting the program is the potential increase in speed, energy efficiency and cost effectiveness that could be provided by moving BUDE onto modern parallel hardware. BUDE has already been ported to use high speed hardware acceleration and is currently run on ClearSpeed CSX600 processors capable of up to 50 double-precision GFLOPS (GFLOPS = Giga Floating Point Operations Per Second); this accelerator was approximately 10 times faster than a dual core CPU of its era. This previous port achieved a 10 times speed up over the original relatively un-optimised FORTRAN version. The latest GPU architectures such as Nvidia's Fermi are capable of up to 1288 single-precision GFLOPS. By porting BUDE to run on GPUs another performance increase of up of 10 times or more could potentially be achieved. OpenCL is a cross platform language will be supported on an even wider range of hardware in the future [27]. This should allow the OpenCL version of BUDE to be moved to future faster hardware without any code modification and is demonstrated by it running on a GTX280 and a GTX480 equivalent the C2050.

The secondary aim of the project was to test the program on multiple hardware platforms, including ATI and Nvidia OpenCL-capable GPUs and modern multicore x86-64 processors from Intel and AMD. These tests were performed to find out which platform offers the best absolute performance, performance per watt, performance per dollar, kilowatt hours used per experiment run and C02 emission per experiment run. This not only demonstrates the most cost and power efficient hardware platform for BUDE but also for other real world OpenCL-based programs.

The objectives to meet these aims can be broken down as follows:

1. **Implement a version of BUDE in OpenCL that works on x86 CPUs.** Start off with a relatively simple un-optimised version of BUDE in OpenCL. Only needs to work with x86 CPUs initially.
2. **Make necessary modifications to allow the OpenCL kernel to run on GPUs.** This involves changing where memory is allocated so it can fit in a GPU's smaller memory. Should run on NVIDIA Fermi and GTX280 GPUs as a priority as these will be used by the university.
3. **Optimise the program to improve performance.** Optimise the program to make full use of OpenCL's parallelism and deliver the best possible performance. If time permits attempt platform dependant optimisations.
4. **Benchmark program on x86 CPU, GPU and ClearSpeed accelerator platforms.** Compare all existing versions of BUDE including the original Fortran, ClearSpeed and new OpenCL versions. Benchmark on as wide a range of hardware as is possible.

Compare and contrast the performance, performance per watt, performance per dollar, cost per run and carbon emissions per run to identify the most efficient platform.

6 Background review

6.1 Bristol University Docking Engine

Bristol University Docking Engine (BUDE) is a drug docking algorithm developed at Bristol University. It ranks a library of protease inhibitors (drugs used to treat or prevent viruses) in terms of their predicted binding affinity using the Evolutionary Monte Carlo (EMC) search method. BUDE is given two molecules, one protein (drug target) and one ligand (protease inhibitor), with which it performs tests to see how well they fit or 'dock' for different poses (different positions of the ligand relative to the protein). Figure 1 illustrates a good fit. The ligand can be either another protein, a peptide or other drug-like molecule. The simulated strength of the drug is measured by a parameter called 'forcefield' which is produced by formulae using the spatial properties and characteristics of the protein and ligand being docked. The forcefield concept is unique to BUDE and distinguishes it from other similar drug docking software. Strength of the forcefield is determined by doing individual calculations between every atom inside the protein and every atom inside the ligand; a low calculated energy means there is a high probability the protein and ligand will fit. Out of a set of results it is the interactions that produce the lowest forcefield that would make the most effective drugs. The theoretical binding energy for each pose can then be ranked by predicted energy [1].

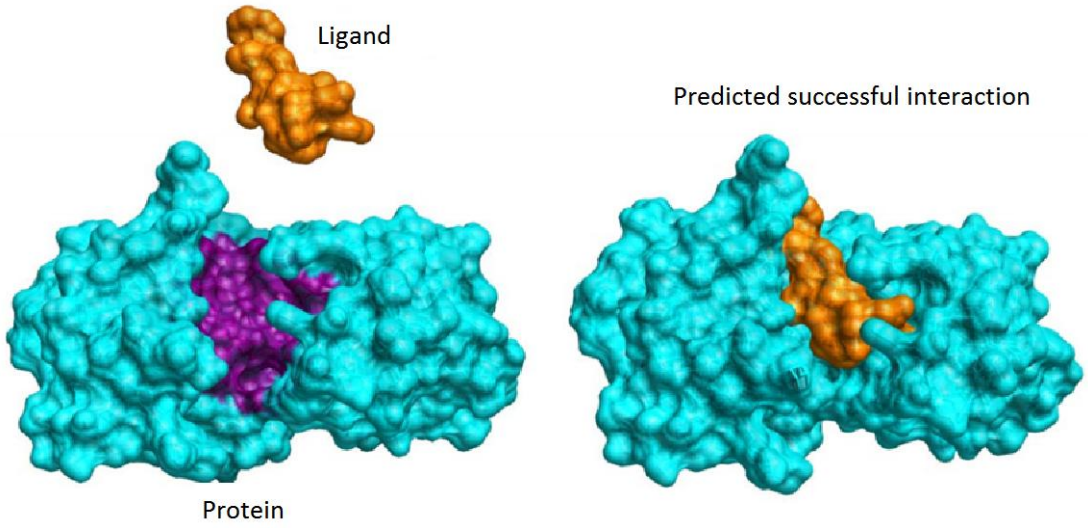

Figure 1, interaction between enzyme and peptide. Image from [3].

Using a computer to simulate drug docking is cheaper and faster than laboratory testing which allows more protease inhibitors to be tested for the same amount of time and money. However BUDE is not intended to completely replace laboratory testing, it simply identifies protein ligand combinations that are likely to be successful as a drug for further, more in depth, testing in labs. Although BUDE is fast when compared to doing the same tests in a lab it does require vast amounts of computing power due to the number of calculations that must be done for each

experiment. More computing power means that experiments involving larger molecules, such as protein-protein can be done, or larger numbers of potential drugs candidates can be tested simultaneously which raises the chance of a successful drug being found.

To carry out a docking experiment BUDE moves the ligand on a 3D grid based around the centre of mass of the protein. The program will move the ligand around, based on its own centre of mass spatially along three axis' X, Y and Z. The ligand can also be rotated on three axis' Xr, Yr and Zr. The combined 6 degrees of movement are bunched together in a number called the pose descriptor which describes the ligand's position, or pose, for that particular experiment.

The program is given a protein and ligand to test. The program reads the test instructions from a file then assigns parameters to the protein and ligand accordingly. It then generates a set of pose descriptors for the given protein and ligand which describe the position of the ligand relative to the protein. These descriptors are generated at random by the program to attempt to sample a wide range of possible orientations, or poses. The user has some control over the formation of the poses to be generated. The larger the value the finer the granularity of the test and the longer it will take to run. The precise number of this range can be varied by the user, as can the physical area the range maps to. If the user knows the ligand is likely to bind better on one side than the other, the rotation can be restricted so only one side is tested. The pose descriptors are then turned into transformation matrices which are used to apply the new pose of the ligand from the original one.

After these steps have been completed the EMC part of the program begins. This part of the program takes up more than 99% of the processing time due to the large numbers of calculations that must be performed. A typical protein is between 1000 and 10,000 atoms and a typical ligand is between 100 and 500 atoms which means an average energy calculation will involve 1.5 million smaller calculations which themselves are split into multiple parts with various constraints that need applying. For a protein-protein experiment this number can be much larger, involving two 10,000 atom molecules. However there is a maximum range over which the interactions between individual atoms are appreciable and due to the size of proteins many atoms will be beyond this maximum range. This reduces the number of calculations necessary in protein-protein situations but range checks must still be done for each atom.

After the EMC searching and energy calculations are complete the program will end. The precise point at which the program stops can be set by the user. After each iteration the pose descriptor with the lowest energy level is saved. As the program finishes all the pose descriptors with the lowest energy levels are compared. The results will be saved and the user will be informed if any of the tests produced energy levels that would indicate the ligand is suitable for further testing in a laboratory.

6.1.1 EMC search method

BUDE's task when doing energy calculations is to search for the pose with the lowest energy out of all the poses it tests. The program can be set up to exhaustively try every possible ligand pose

but the number of calculations and therefore computational power necessary to do this would be so enormous it would be impractical. As an example the GA (genetic algorithm) test used for benchmarking in this project will calculate the energies for approximately 1% of the entire search space to find a pose reasonably close to the best possible. Doing an exhaustive search would involve calculating energies for 100% of the search space. Research into genetic algorithms is still on-going and beyond the scope of this project.

The Evolutionary Monte Carlo (EMC) search method, originally applied to protein folding [2], is used by BUDE when performing a GA iteration. EMC samples only a subset of possible poses, significantly reducing the number of calculations, but still locates low energy minima.

After the first pose descriptors have been generated the energy levels of each pose are calculated. After the calculations for all the generated pose descriptors are complete the results will be ordered by strength of forcefield and the pose descriptors with the lowest forcefield will be saved. The group of saved pose descriptors, known as parents, will then be used to generate a new set, known as children, which are based on the parents. The new set of child pose descriptors then go through the same forcefield energy calculation process and again generate new children based off the ones with the lowest forcefield levels. This iterative process is repeated until a satisfactory range of possible energy levels have been generated. See Figure 3 for an example of this.

This is done by limiting the random generation to certain parts of the pose descriptor. Table 1 shows an example of how iterations are performed in the BUDE program. In this case X and Y and locked so the child descriptor shares the same X and Y coordinates as its previously generated parent pose descriptor. Because 2 out of 6 axis are locked this is called a mutation rate of 33%. All the other pose descriptor coordinates are free to be randomly generated.

Table 1, child and parent pose descriptors with a mutation rate of 33%

Parent pose descriptor:	X	Y	Z	X rotation	Y rotation	Z rotation
	1	4	6	6	6	6

Child pose descriptor:	X	Y	Z	X rotation	Y rotation	Z rotation
	1	4	2	3	3	0

Another alternate method which could be used is limiting the generation of new coordinates in children to be within XY% of their parents. This would shuffle the search area around until the minimum is found similar to how tests would be done in a lab. Using the example shown in Table 1 this would mean child descriptor would allow all axes to be modified but only within +/- 33% of the values of the parent. EMC is used instead because of the minimisation problem (also known as the optimisation problem). Figure 2 shows an output for a set of pose descriptors after the energy levels have been calculated. If the starting parent was pose 13 the alternate shuffling method would slowly move towards pose 12 until the minima has been found. It would then get stuck in a local minima and the program would presume the lowest minimum has been found. However it can be clearly seen on the graph that the minimum is between poses 7 and 8. The shuffling method results in the program getting stuck in local minima which

can result in it missing better results. The method used in BUDE results in a scattering of results over a wide area. Though this seems unfocused it produces a more complete analysis of all the minima in cases where the results produce a very rough surface with many minima. Once the principal minima have been found the alternate shuffling method could be used to find the lowest point, though BUDE is currently set to use EMC throughout.

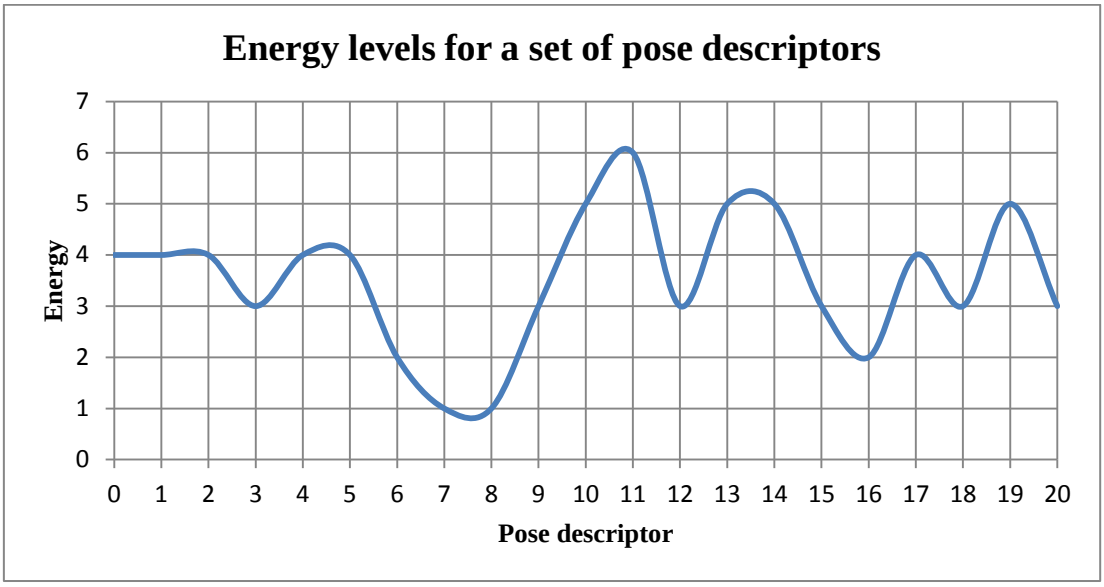

Figure 2, Energy levels with false minima

The new set of child pose descriptors then go through the same forcefield energy calculation process and again generate new children based off the ones with the lowest forcefield levels, see Figure 3. This iterative process is then repeated until a satisfactory range of possible energy levels have been generated, as defined by the user.

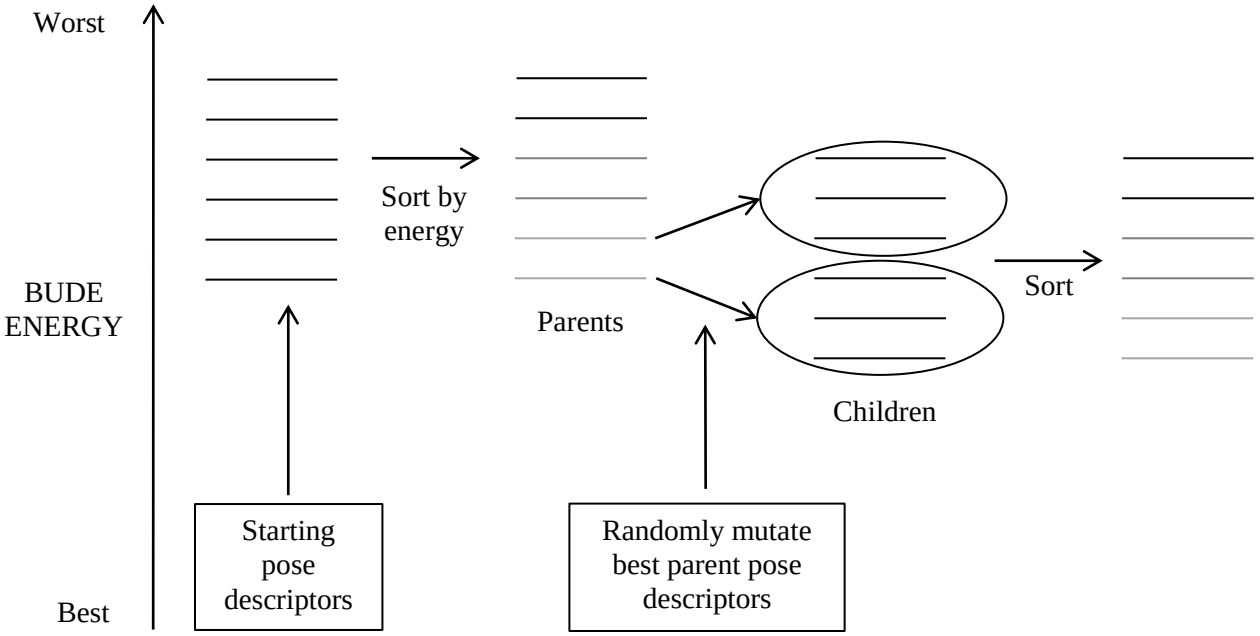

Figure 3, EMC search method used by BUDE

6.1.2 BUDE on ClearSpeed

BUDE was initially developed in the FORTRAN programming language. This severely limited the speed at which a single instance of program would run because any one instance of the program could only make use of a single CPU core. BUDE can use MPI to allow multiple instances of BUDE to be run in an attempt to use all available CPU cores but this does not increase the speed of an individual instance. To provide more compute power BUDE was ported to the ClearSpeed accelerator platform using its Cn parallel language [3]. The ClearSpeed CSX600 processor add-in card is the model currently used for BUDE. The processor is designed as a math co-processor capable of 50 GFLOPS double precision at 5 watts power consumption [4]. This is compared to a CPU processor from the same period; an Athlon 64 X2 4600 managing only 6.93 GFLOPS with 90 watt power consumption for both cores or 3.46 GFLOPS for one core using Linpack. The benchmark used to calculate the GFLOPS of the CPU made use of SIMD extensions that BUDE is unable to make use of so the available CPU processing power for BUDE would be lower. The ClearSpeed card has other advantages as well as increased compute compared to a CPU including a far larger memory bandwidth at 96 gigabytes/s bandwidth to on-chip memory compared to 4-5 gigabytes/s for an Athlon 64 based processor. This results in the Cn ClearSpeed version of BUDE being approximately 10 times faster than the FORTRAN version. Each ClearSpeed accelerator card includes two CSX600 processors, and because of their

low power consumption, multiple ClearSpeed cards can be installed inside one computer. BUDE is able to use these multiple ClearSpeed cards simultaneously which means it can make use of the massive parallelism available and increase the speedup even further.

The Cn version of the program works slightly differently to the original FORTRAN code due to the extra piece of hardware involved. Figure 4 shows a flow chart of BUDE implemented on ClearSpeed. Over 99% of the processing is done on the ClearSpeed half of the program. The CPU host side is still implemented in FORTRAN. The main difference is that after the parameters are assigned they are copied over to the Clearspeed card. The CPU host then generates the Pose Descriptors (PD) and turns them into Transformation Matrices (TM). The TM's are then sent to the ClearSpeed card where it applies them to the data within its memory. This takes very little time due to BUDE only requiring small amounts of data (40 bytes per atom), and this moved infrequently. The majority of time is spent on the next stage where the energy calculations are performed. After the calculations are complete the resulting energy levels along with their relevant PD's are sent back to the host side of the program which will then repeat the process for N generations. A similar setup is used in the OpenCL version of BUDE.

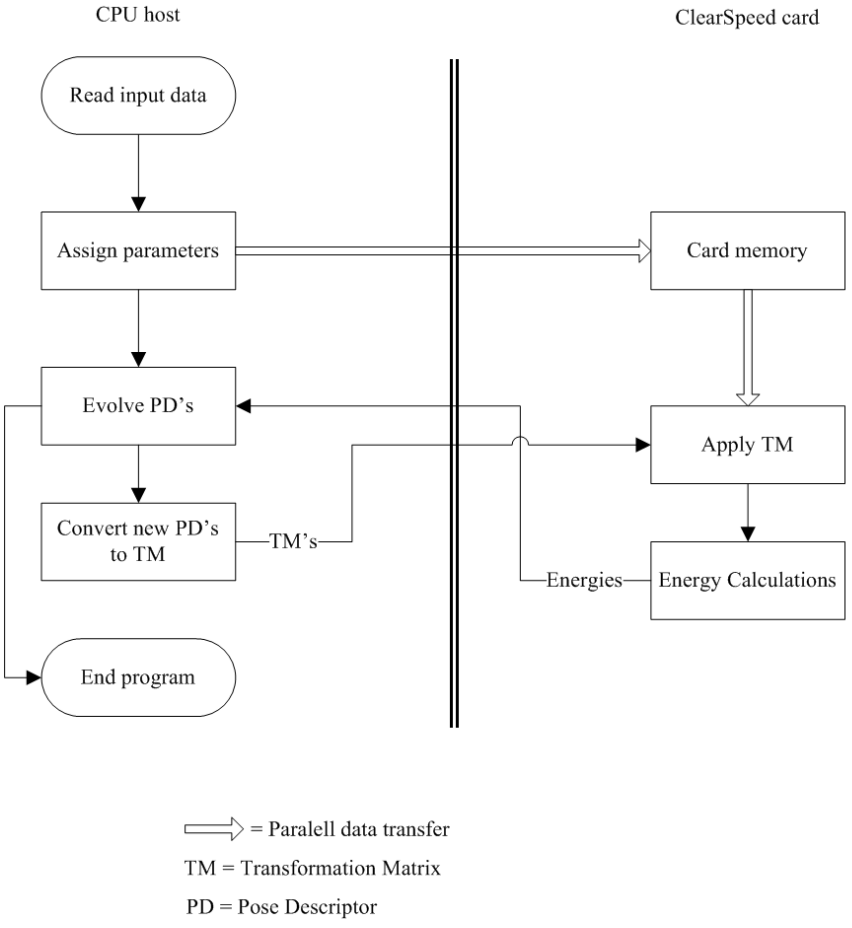

Figure 4, BUDE implemented on ClearSpeed

6.2 High-Performance Computing

Drug docking applications like BUDE require High Performance Computing (HPC). Traditionally if you needed a HPC system to run a compute intensive application you would have to either purchase a supercomputer or pay for time on one. Up until very recently these supercomputers used a wide variety of different architectures. Generally this involved large numbers of special purpose CPU type processors such as IBM's POWER series, Sun Microsystem's SPARC, Alpha DEC and Intel Itanium. However since late 2000 there has been an explosion in the use of normal desktop x86 processors such as Intel Xeon and AMD Opteron [5]. The Top500 HPC statistics show that in Nov 2000 only 1.2% of the top 500 HPC systems were based on x86; by November 2008 this had risen to 85.8% [23]. Although many more of these normal processors are necessary for the same processing power as the more specialised architecture they are far cheaper to acquire due to them already being mass manufactured for use in desktop computers. This makes the whole supercomputer cheaper for the same speed with x86 processors. The market is now so big that Intel has started to design its consumer CPU architecture with HPC and servers in mind such as with its new Nehalem architecture.

6.2.1 Anton custom architecture

Another option when looking at HPC for MD (Molecular Docking) is a custom architecture designed specifically for MD such as D.E.Shaw's Anton [7]. Anton is made up of a substantial number of Application Specific Integrated Circuits (ASICs) design to run MD software such as GROMACs [8]. GROMACs is a widely used piece of MD software with unique processing requirements. The various ASICs in Anton are designed to perform these specific parts of a GROMACs type program as fast as possible. The ASICs are then linked together via a specially designed high speed 3D torus network. This means that there should be few if any bottlenecks in the processor which allow it to run the software it is designed for extremely fast. A general purpose processor is capable of achieving simulation rates of a 300-400 nanoseconds of protein-water interactions over 24 hours while Anton is estimated to achieve 10,000 nanosecond in 25 hours [8][9]. This results in an increase in simulation speed of approximately 25 times. Normally creating a processor such as this would not be attempted due to the rapid rate at which general purpose processors advance. It could be possible for a specialised design to be created only to find it superseded by general purpose processors a couple of years after it rolls out. This possibility is made more likely because general purpose processors such as Intel's tend to be manufactured in the most advanced fabrication facilities in the world whereas custom ASICs such as Anton have to make do on previous generation facilities. The funding required to create an ASIC such as Anton is also vast in comparison to using an off-the-shelf general purpose processor. Anton was funded entirely by its creator D.E.Shaw who has the intention of using it to make scientific breakthroughs. This kind of funding is well beyond the reach of most MD projects including BUDE.

6.2.2 GPGPU

As high performance CPUs become more parallel and focus less on single thread performance the reverse has started to happen with some of the most superscalar processors around, Graphics Processing Units (GPUs). GPU architectures are nearly always massively superscalar and parallel; they are designed for high throughput rather than concentrating on the performance of a single thread. They concentrate on extremely high aggregate performance and consequently the die of a GPU is focused on arithmetic units rather than cache. GPUs used to be so focused on arithmetic computation that anything but graphics processing was close to impossible on them due to a limited inflexible range of instructions and very large performance penalties for stalled threads. In addition to this they used custom unstandardised methods of doing calculations such as non IEEE754 floating point because the only software that would ever need to interact with the hardware was the GPU's driver. More recently GPU manufacturers such as Nvidia and ATI have increased the complexity and capability of the cores on their GPUs by increasing the number of instructions available and adding small caches. This means modern GPUs such as Nvidia's G80, GT200 and Fermi designs as well as ATI's RV770 and Evergreen designs are able to be effectively used as general purpose processors as well as for graphics. GPUs used for general purpose programming instead of just graphics are known as General Purpose Graphics Processing Units (GPGPUs).

These new GPGPUs are becoming popular for scientific compute intensive use due to their enormous processing power, low cost and wide availability. In particular they are being used for Molecular Docking [10]. This allows scientists who are unable to access or use HPC clusters to employ new simulation and analysis techniques. The latest GPGPU capable GPU is Nvidia's Fermi. It has 480 cores and is capable of 1 teraflop single precision or 500 gigaflops double precision with an on card memory bandwidth of 173 gigabytes per second. Not only does the card have a huge amount of raw processing power available it has been designed with HPC use in mind.

Fermi's architecture contains many special features and improvements over previous generation Nvidia GPGPUs and its competitor ATI's GPGPUs. Fermi is massively multithreaded to hide memory latency which speeds up processing. To ensure thread context switches don't stall the system Fermi has many registers which store the context and can rapidly restore it. Fermi groups data parallel threads together into 'warps'. Thanks to dual issue stream processors (clusters of simpler cores) multiple warps can be scheduled to run simultaneously further increasing performance in situations where there are no data dependencies. Fermi has full ECC-SECDED (Single Error Correct, Double Error Detect) on all memories; this is the only GPU available with ECC to date. Fermi comes with a configurable L1, L2 cache and high speed low latency GDDR5 RAM. Multiple kernels can be run in parallel on Fermi (kernels are themselves parallel programs), another feature which is missing from ATI and previous generation Nvidia GPUs. Fermi has the latest IEEE 754-2008 support which includes support for fused multiply-add and subnormal numbers.

Table 2, specifications of potential platforms for BUDE including the existing ClearSpeed platform.

	Nvidia Tesla C2070 'Fermi'	AMD ATI FirePro V8800	Intel Nehalem Xeon X5680	Clearspeed CSX600
Price (ex VAT)	~£2,800	~£1,000	~£1,200	N/A already purchased
Core clock	700Mhz	825Mhz	3330Mhz	250Mhz
Cores	448	1600 (320 double precision)	6 (12 with SMT)	96
RAM memory	6GB	2GB	user configurable	up to 4GB
Single precision (GFLOPS)	1000	2640	~200	50
Double precision (GFLOPS)	515	528	100	50
Memory bandwidth	173 GB/s	153.6 GB/s	30 GB/s	96 GB/s
Power consumption under load	250W	188W	130W	5W
Transistors (millions)	3000	1500	1170	128
Full ECC	yes	no	yes	yes

Table 2 shows the hardware platform BUDE is currently running on and platforms it could potentially run on once ported to OpenCL. The components selected are part of the manufacturers' professional line designed with use as a GPGPU in mind. It would be possible to use the significantly cheaper consumer graphics card variants which are equally performant but often have less RAM, are physically slightly larger and offer less support for GPGPU usage [16]. From this table I would predict BUDE to run at least ten times faster in OpenCL on Fermi than it currently does on a single ClearSpeed CXS600.

The ATI FirePro V8800 initially appears to be highly competitive with the Nvidia Fermi based C2070 however it lacks ECC memory. This is valuable feature for a scientific program like BUDE. If an error occurs and is undetected an experiment could produce drastically incorrect results. This could end with money and time being wasted on laboratory test or a potentially useful drug being dismissed as useless. To counter this, error checking could be implemented in software but the performance reduction due to software error checking is often large, halving the performance or worse and increasing memory requirements on the card (V8800) that has the least memory available. The biggest advantage of Fermi over the ATI card is one that cannot be seen on the spec sheet in table 2 which is the way the architecture is designed. As mentioned earlier Fermi is designed specifically with GPGPU in mind while the ATI V8800 was

designed with 3D consumer graphics as the main market with GPGPU stuck on after. Figure 5 shows a benchmark performed by Anandtech. The GTX 480 is the consumer variant of Fermi and the Radeon 5870 is the consumer variant of the FirePro V8800.

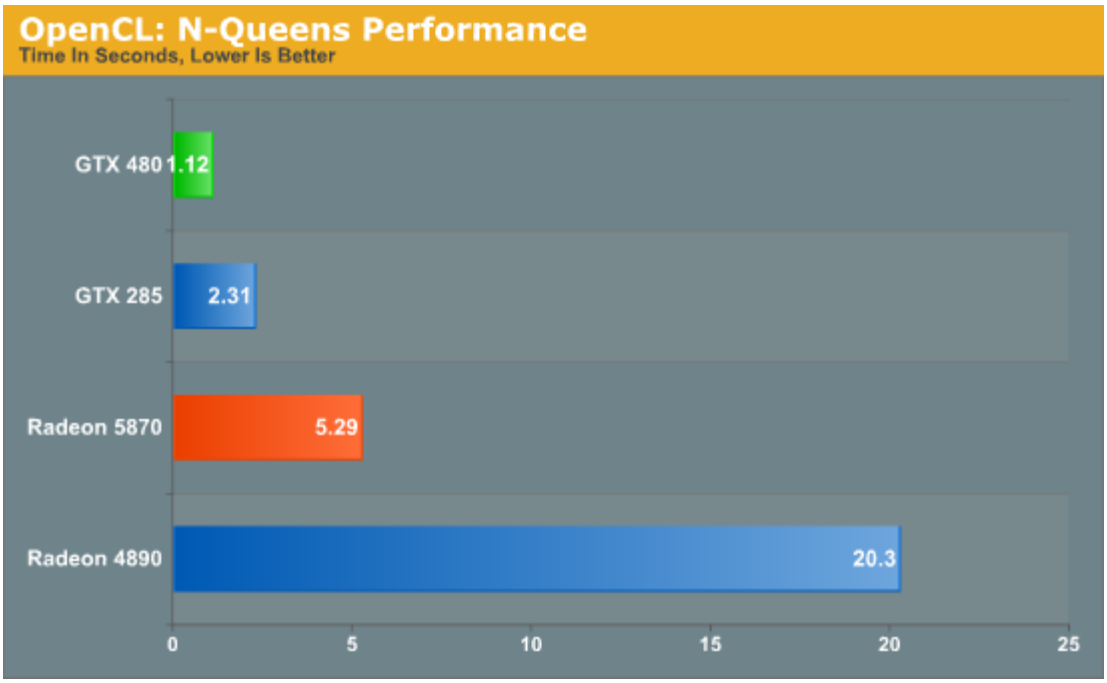

Figure 5, OpenCL benchmark sourced from Anandtech [15]

Even though the V8800 looks competitive with Fermi on paper in real world tests Fermi vastly outperforms it due to having superior thread management and caching. In this case Fermi is nearly 5 times faster than the V8800. Going by this benchmark I would predict a Fermi based card to outperform an ATI Evergreen (V8800) based card for BUDE. However, bearing in mind that this program, though it is an OpenCL program, simply demonstrates how well the cards perform for the N-Queens algorithm it is possible BUDE is better suited to ATI's architecture.

6.3 Parallel programming languages

In the past CPU microprocessors drove forward rapid increases in performance and reductions in cost, scaling up with transistors from Moore's law. Thanks to innovation from Intel this was accomplished by making single cores ever faster and ever more complex. In the 80's the first attempts at multicore processors were made to increase performance without increasing complexity of the cores. Although the theoretical speeds of these processors were good the big problem was writing programs for them. Intel quickly designed and produced single core processors that offered equal performance to these multicore designs but ran existing sequential software. Intel's advance temporarily stopped further development of general purpose multicore processors. However more recently the complexity of individual cores has levelled off and even Intel is now moving to multiple cores. GPU's have been moving in this direction for a long time but until recently the cores were too simple to be of any general purpose use. Just as in the 80's these new multicore processors offer great theoretical performance, but actually making use of the raw power through software is challenging. This means software developers can no longer rely on hardware to drive forward performance; substantial software design effort must be made to make full use of existing hardware to gain maximum performance.

By far the most powerful of all widely available low cost general purpose processors are GPUs. They are approximately an order of magnitude faster than equivalent multicore CPUs and have been growing in speed at a faster rate [17]. Other high performance processors include IBM/Sony's cell.

Many parallel programming languages have been suggested over the past few decades to make use of parallel processors. Until 2006 GPUs were very difficult to program because the only languages available were intended for graphics programming such as OpenGL or Direct3D. These languages required highly specialised knowledge and did not provide many of the functions necessary for normal programming due to limited APIs underneath. There have been earlier attempts at GPGPU languages such as Brook and Cg. Brook which used DirectX 9+ or OpenGL 1.3+ for its backend was the most successful. This use of DirectX and OpenGL allowed Brook to run on many platforms but exposed it to the same limitations. Even with this limitation there have been a few hugely successful projects, such as the Brook based Folding@Home [21], a protein folding distributed computing project which can is capable of running on older GPU architecture such as ATI R520 'Fudo' GPUs. Folding@Home has since moved to Cuda and OpenCL.

6.3.1 CUDA

In 2006 Nvidia released its CUDA parallel programming language along with a new GPU architecture called the G80. The hardware itself had extra area in silicon added to accommodate CUDA features designed to facilitate easier parallel programming. Programs

using CUDA no longer go through the graphics API; instead there is a new dedicated general-purpose parallel programming interface to serve them. At the same time as implementing this new interface with the hardware Nvidia changed higher layers so that familiar C/C++ programming tools can be used. CUDA is widely considered a significant step forward in parallel programming and drug docking software has been written and successfully run on Nvidia GPUs using CUDA [11].

6.3.2 OpenCL

After the success of CUDA other GPGPU parallel programming languages have been released. AMD (ATI) released its Stream platform including the CAL language which works much like CUDA but on ATI cards. Due to being late to market and ATI putting less effort into HPC on GPUs, Stream has met with little success so far. Both CUDA and CAL rely on Nvidia and ATI respectively to continue providing support for the future and both only work on hardware from their respective company. To fix this problem a new open source language called Open Computing Language (OpenCL) was released in August 2009 [20]. OpenCL is a collaborative project between a large number of major industry players including Apple, Intel, AMD (ATI), Nvidia and IBM who have formed the Khronos group consortium. Due to the wide range of partners OpenCL can run on a wide range of platforms; this alone makes it highly likely OpenCL will be the most popular GPGPU language in the future. OpenCL is not just limited to GPUs; it can also run on general-purpose x86 CPUs and make full use of advanced SIMD instructions to get maximum performance. It will also work with a variety of other processors including IBM/Sony's Cell and Intel's future Larrabee CPU/GPU hybrid architecture. Not only is OpenCL able to run programs on a wide range of hardware it can use different pieces of hardware simultaneously. It is possible for a single instance of an OpenCL program to make use of a multicore x86 CPU and a GPU simultaneously by splitting serial workloads to a CPU and data parallel workloads to a GPU (see Figure 6).

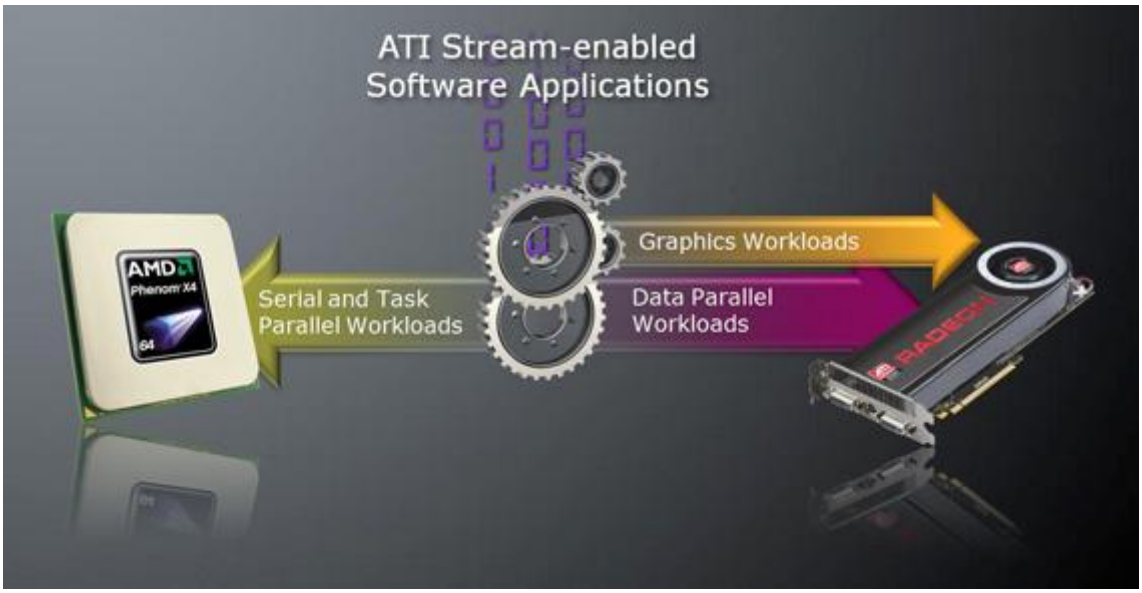

Figure 6, splitting of workloads in OpenCL. Image from AMD OpenCL website [22]

When running on Nvidia and ATI hardware OpenCL goes through the hardware's respective CUDA and Stream APIs. Similar to CUDA OpenCL provides programmers with language extensions and runtime APIs to allow management of parallelism and data delivery. Because OpenCL is a standardised language applications developed in it can run without modification on all platforms. However platform set up commands that run at the beginning of the program may still need modifications for each platform. In terms of programming, OpenCL is a slightly lower level language than CUDA with both being based on C. Due to OpenCL being open source there are now other versions of OpenCL based off different languages such as C++.

Although OpenCL is in its infancy its wide hardware and operating system software compatibility makes it the most suitable language to use in enabling BUDE to make use of modern parallel GPU hardware. Presuming no significant changes are made to the OpenCL standard it should be possible to move an OpenCL version of BUDE onto newer hardware when it arrives.

6.3.3 DirectCompute

Microsoft decided to stay out of the Khronos group and create its own parallel programming language called DirectCompute (along the lines of the successful Direct3D). Though DirectCompute works across a range of hardware it will only run on Microsoft Windows Vista and Windows 7 operating systems which limits its appeal. It runs on any GPU capable of DirectX10 or DirectX11 and shares computational interfaces with both CUDA and OpenCL.

6.3.4 MPI

Another parallel programming API that has found success in the scientific computing domain is Message Passing Interface (MPI). This API is designed for scalable cluster computing and is normally used with OpenMP for shared memory multiprocessor systems. In MPI clustered computing, nodes do not share memory; all data sharing and any interactions must be done through explicit message passing. Applications written in MPI have successfully been used on clusters of over 100,000 cores [19]. However the amount of effort required to program in MPI is extremely large due to the lack of shared memory. When OpenMP is used there is support for shared memory which reduces programming effort; however OpenMP has failed to scale beyond more than a couple of hundred cores due to problems with overheads for thread management and cache synchronisation. CUDA, OpenCL and DirectCompute support high scalability with low overhead thread management and no cache coherency issues.

6.4 Background research conclusions

The porting of BUDE to ClearSpeed established that there are great benefits to be gained from moving BUDE to high performance parallel software and hardware platforms. Other programs ported onto GPUs show the potential for significant realistic speedups of 5-10X compared to using high-end CPUs [6]. Both of these clearly demonstrate the potential of a GPU port of BUDE. The ability of OpenCL to work with a wide variety of operating systems, including Linux, Windows and OSX and with a wide range of hardware including the most popular GPUs and CPUs gives it a strong advantage over other languages. The CPU and GPU industries are both moving towards a merging point where a single processor is capable of both high performance graphics and general purpose programming, notably starting with AMD's 'fusion' APU [25]. OpenCL is well placed to make full use of these new hybrid processors which should ensure forward compatibility. In addition the ability to use different hardware for different tasks internally within OpenCL should allow maximum performance to be extracted from a system by balancing the data parallel and task parallel parts on the GPU and CPU respectively. The wide range of support from manufacturers makes it likely the future of OpenCL is secure unlike CUDA which is dependent on Nvidia's continued investment. OpenCL is clearly the correct choice for a new version of BUDE. The CPU/GPU split in the new version of BUDE should initially be similar to the ClearSpeed version.

7 Project design

The existing version of BUDE has timers built into it to identify which parts of the program are taking up the most compute time. For the original scalar FORTRAN version the timing results are as follows:

```
TIMING                                  user      system
======                                  ====      ======
Time spent calculating geometry:         0.6 %        0.0 %
Time spent calculating energy:          99.2 %        0.0 %
Time spent calculating RMS:              0.0 %        0.0 %
Time spent sorting conformers:           0.0 %        0.0 %
Time spent evolving conformers:          0.0 %        0.0 %
```

The 99.2% of the time is spent calculating energies so, just like the previous ClearSpeed accelerated version, this is the area of the program that will be targeted for speed improvement. The calgen() routine is responsible for differentiating the different types of energy calculation necessary and setting them up. The only energy calculations which take up a notable amount of time are the ones performed by the fasten() routine. In the ClearSpeed version fasten() was converted into ClearSpeed's Cn code and run on a CSX600 accelerator. An extra file was added which contained functions calgen() could call in order to control the ClearSpeed accelerator. The OpenCL version of BUDE created in this project is based off the previous ClearSpeed work and works in a similar fashion.

Figure 7 shows a simplified flow diagram of the OpenCL version of BUDE. The only difference in terms of data flow compared with the ClearSpeed version in figure 4 is the addition of a more complex OpenCL interface. The OpenCL device side is purely the fasten() routine and, if a CPU OpenCL device is selected, this can run on the same physical processor as the host code.

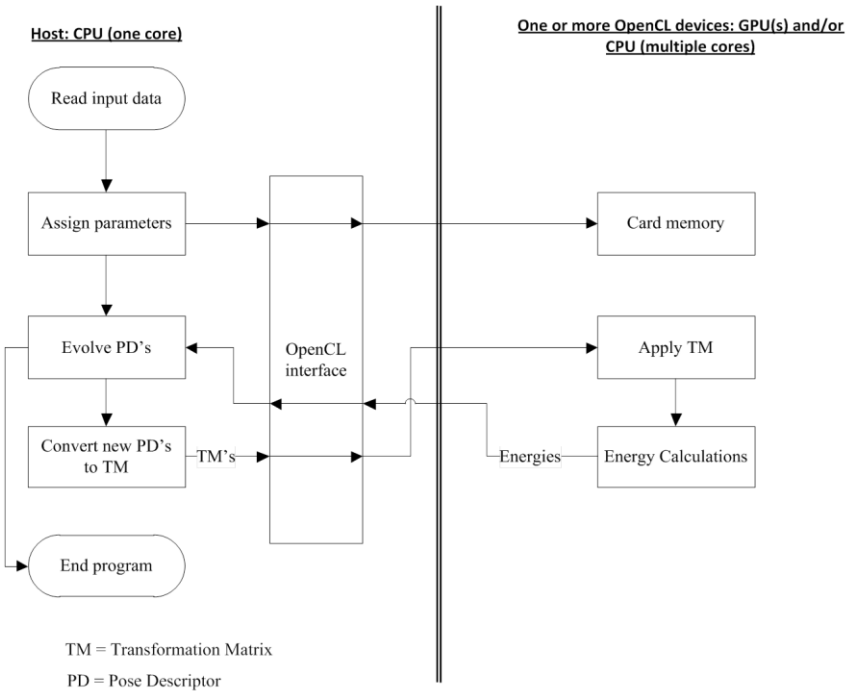

Figure 7, simplified flow diagram of BUDE with OpenCL acceleration

The following files were either modified or added to the BUDE program for this project:

Host code:

- Makefile
- bude_cl_calgen_rot.f
- bude_cl_interface.c
- bude_cl_h.h
- bude_cl_host.cpp
- bude.f

Kernel code:

- docking_compute.cl

All of the above can be found in the appendices.

7.1 Host code

In order to facilitate a version of the fasten() routine on an accelerator the ClearSpeed version came with an extra C file that acted as an API allowing FORTRAN code to control the fasten() routine. A similar setup was used for the OpenCL version. Although OpenCL's API could be used directly from FORTRAN by creating a wrapper for the vendor provided header files it would involve a disproportionate amount of work for any benefit that might come of it. There are many wrappers available but at the time the program was being developed there was no FORTRAN version so a C language host program is used as a translator between FORTRAN and

the OpenCL API. This will also make it easier for future work when the main BUDE host code is converted from FORTRAN to C++ (see chapter 10).

The C OpenCL host program contains all of the OpenCL setup API calls inside in order to simplify the calls made from FORTRAN as much as possible. The OpenCL host code is contained in a new file called bude_cl_host.cpp. Although it has a .cpp extension it is actually written in C style, this was necessary due to a quirk in the vendor provided OpenCL API header files.

The OpenCL host in bude_cl_host.cpp is a normal OpenCL host program split into functions so they can be called from the calgen() routine, implemented in FORTRAN. There is an extra C file inbetween the CL host functions and the calgen() FORTRAN code along with a header file to facilitate. These two files could be consolidated into bude_cl_host.cpp and were only created separately for clarity and flexibility.

For detail on what the various functions in the OpenCL host file do see the code comments in bude_cl_host.cpp found in the appendix.

The only other FORTRAN file to be modified in this project was bude.f, where two OpenCL host calls are made. The calls added are cl_init() and cl_finalize() which allocate and setup then free the necessary variables for use with the OpenCL API. In the ClearSpeed version equivalent functions were performed inside the calgen() routine which resulted in unnecessary repetition and time wasted on re-initialising variables, moving them to the main bude.f file means they are only called once each time the program is run.

One of the most time consuming parts of the project was getting both the original version of BUDE to compile and a version that could also run OpenCL code. A significant amount of time was spent getting the original to compile due to Fortran77 being obsoleted from all modern operating systems. Unfortunately attempts to get it working on Microsoft Windows failed entirely. Even the original version of BUDE requires Fortran77 and C code to be compiled alongside each other. Although there are Fortran77 and C compilers available for Windows, none were found that could do both. The university computers than run BUDE had been set up 4 years ago and had a multitude of old software installed on them providing compatibility. Discovering what programs and libraries were necessary to run BUDE on a new install of Linux took a significant amount of time and investigation. To run the original version of BUDE it is necessary to have both f77/g77 and gcc. The former is obsolete and has been replaced by gfortran which produces non-trivial and still unresolved linking errors when attempting to compile BUDE. The addition of OpenCL into the mix created even more problems. OpenCL requires the C++ compiler g++ to compile. This means extra libraries (eventually identified as libf2c and libg2c) are needed to enable g++ to compile Fortran77 code.

The OpenCL host code is the most complex part of developing an OpenCL program due to the large number of API calls necessary to perform simple tasks such as passing an argument to a function. If you want to pass an argument to the OpenCL kernel that is an array you must first (after initialising the many required variables) create a memory object, enqueue the write buffer with the memory object and the object to be written, then set the argument. There are

wrappers available that automate a lot of it and allow OpenCL to be run from other programming languages, such as a Java wrapper and a C++ wrapper, but these add further Fortran77 compatibility complications and there is less support available.

After the initial versions of the host code were produced further optimisations were necessary to increase performance. In the host code the way in which kernels are queued up was modified to ensure as many work items as possible are queued up awaiting execution. This optimisation made no discernable difference to the execution time of BUDE on CPU OpenCL devices but resulted in improvements of approximately 20% on the fastest GPUs such as Nvidia's GTX280. Originally the OpenCL host code queued up 40 times the max_workgroup_size (as reported by the clGetDeviceInfo API call) in batches until the total number of transforms/poses remaining was less than 40 x max_workgroup_size. The number 40 was used to ensure the GPU has a big enough workload without overloading its memory and it was multiplied by max_workgroup_size so it scales up with faster GPUs. After this it would queue up a single work group with all the remaining poses. When running on a GPU like the GTX280 this meant only a single group of cores, or 'SM', was used meaning 232 out of a total of 240 cores were idle. This resulted in a significant pause when computing the remaining transforms/poses slowing down the overall execution time. To speed this up the way transforms/poses are distributed was changed so the maximum number of work groups are being used as much as possible, this can be seen at lines 429 to 453 of the bude_cl_host.cpp.

7.2 Kernel code

The OpenCL kernel is very close to the ClearSpeed Cn implementation of the fasten() routine. Like the Cn version it replaces the Fortran implementation of fasten() and is called from the Fortran calgen() routine. The main energy calculation code seen in lines 149 to 238 of docking_compute.cl are almost identical apart from some OpenCL related pointer changes. Both OpenCL and ClearSpeed achieve best performance if there are as few data dependencies as possible which makes BUDE ideal. However the way in which the kernel is called in OpenCL and ClearSpeed is different which results in differences to both the OpenCL calls made from the calgen() routine and the loops inside the OpenCL kernel version of the fasten() routine.

In ClearSpeed the Cn file is simultaneously run on all cores of a ClearSpeed processor such as the CSX600. Any variable that is declared as a mono is held in global card memory (RAM) meaning it is scalar and a single instance of it exists. When operations are performed using mono variables they are performed serially. A variable declared with poly in front of it is stored locally in each processing element of the accelerator. BUDE uses a CXS600 card so there are 96 instances of each poly variable. It can be thought of as a scalar processor with SIMD and giant 96 way vectors.

```
mono int scalar_var;
poly int parallel_var;
```

Mono and poly variables in OpenCL kernels work differently to a ClearSpeed Cn file. OpenCL kernels are scalar (unless vectors are used for certain platforms) and must be queued up to be run multiple times simultaneously to achieve maximum parallel performance. Up to a certain vendor dependant limit the more OpenCL kernels are running simultaneously the faster it will perform. Another way in which OpenCL and ClearSpeed differ is the memory hierarchy on the card. Both separate the on-card memory into multiple parts for the main shared RAM which is accessible by the host and a smaller private memory unique to each processing element. However OpenCL has two extra levels of memory hierarchy compared to ClearSpeed. In OpenCL mono is known as global and poly is known as private. There are then two extra levels that sit in-between global and private, called local and constant. These two extra levels are still shared across all processing elements like global memory but they are inaccessible by the host and two orders of magnitude faster than global memory. In BUDE the arrays stored in shared memory are too large to fit into local or constant memory so these levels of memory remain unused.

The differences between ClearSpeed and OpenCL mean changes have been made to the OpenCL version of fasten().

The most significant change is that each kernel now calculates the energy for a single transform (in the code poses are referred to as transforms). This means if there are 600,000 transforms to be run the kernel will be queued up 600,000 times. If the number of transforms to be calculated is sufficiently high like the previous example this provides a large amount of parallelism allowing for significant speed increases as the number of cores increase. Changing this involved removing the transforms loop from the fasten() routine and modifying the calling code in calgen().

Another change is that the ClearSpeed code can perform up to 4 transforms per processing element at a time. This allowed the code to make use of the ClearSpeed accelerator's per PE SIMD capability. However it was discovered that the performance was often worse when this option was enabled, and hence the ClearSpeed version of BUDE was normally limited to calculating a single transform per iteration of the transforms loop.

7.3 Benchmark methodology

7.3.1 Experiment

The benchmark test used on all platforms searches iteratively using BUDE's genetic algorithm EMC method. It performs 10 iterations running approximately 650,000 energy calculations per iteration. It is tested on all three versions of BUDE, the original pure FORTRAN, ClearSpeed accelerated and the OpenCL accelerated version created for this project.

Benchmarking BUDE involves timing how long a run of the above test takes to run and measuring power consumption of the entire system it is running on. BUDE automatically records the time taken for an experiment and displays it in the main results file. To measure the power consumption of the system a 'Watts Up? Pro' professional power meter was used. This device fits in between the computer's plug and the mains power socket. The meter has a built in data logger and connects to the computer through USB. In this experiment the meter logs an average of the watts used every second. Before logging a test run of BUDE was performed to determine how long the logger would need to be set. Once this was determined the logger was started at least 10 seconds before BUDE starts and left running for at least 30 seconds after to allow for variation in running time and to measure the idle power consumption before each run.

As many background processes as possible were terminated or disabled on the systems while benchmarking was under way. Each test was run five times then the mean average was taken of the five results.

Due to time and hardware availability constraints multiple test rigs had to be used. In ideal circumstances all of the accelerators would have been tested in a single test rig system.

7.3.2 Hardware specifications and test rigs

All systems used 64bit versions of their respective operating systems and 64bit compilers.

All prices are approximations.

Table 3, test rigs used for benchmarking

	Test rig 1	Test rig 2	Test rig 3	Test rig 4	Test rig 5
OS	Cent OS 5	Cent OS 5	RHEL 5	Ubuntu 10.04.1	Ubuntu 10.04.1
Form factor	Desktop	Desktop	Sever blade	Desktop	Laptop
CPU	Intel E8500	Intel i7 920	Intel i7 920	AMD PhenomII X3 720	Intel Core2Duo SU9400
RAM	4GB DDR3	6GB DDR3	6GB DDR3	4GB DDR2	4GB DDR3
GPU / accelerator	Nvidia GTX280	ClearSpeed CXS600	Nvidia Tesla C2050 'Fermi'	N/A	AMD Radeon mobility 4330
GPU driver version	195.36.15	ClearSpeed 3.3	260.21	N/A	Catalyst 10.4
OpenCL/CS SDK/toolkit	CUDA Toolkit 3.0	ClearSpeed 3.3	CUDA Toolkit 3.1	Stream SDK 2.2	Stream SDK 2.1
gcc version	4.1.2 20080704	4.1.2 20080704	4.4.3	4.4.3	4.4.3
Measured Idle	87 W	159 W	Not measureable	76 W	11 W
Cost (GBP)	£900	£4000	Unknown	£450 (2009)	£550 (2009)

For test rig 5 an external display was used and the built in one was disabled. This ensured its power draw was not unfairly increased in comparison to all other test rigs which also did not include displays.

Table 4, compute devices used for benchmarking

	Nvidia GTX280 'GT200'	Nvidia Tesla C2050 'Fermi'	AMD Radeon mobility 4330	ClearSpeed CXS600	AMD Phenom II X3 720 'Heka'	Intel Core2Duo SU9400 'Penryn'	Intel Core2Duo E8500 'Wolfdale'
Processing Elements	240	448	16 (x5 SIMD)	2x96	3	2	2
PE speed	1296 MHz	1150 MHz	450 MHz	210 MHz	2.8 GHz	1.4 GHz	3.16 GHz
Single precision GFLOPS	933	1288	72	80.6	33.7	12.78	28.85
Memory	1 GB	3 GB	512 MB	1 GB	4 GB	4 GB	4 GB
Memory bandwidth	141.7 GB/s	144 GB/s	9.6 GB/s	6.4GB/s	11 GB/s	11 GB/s	11 GB/s
Transistors	1400 Million	3000 Million	242 Million	128 Million	758 Million	410 Million	410 Million
Technology	65 nm	40 nm	55 nm	130 nm	45 nm	45 nm	45 nm
Claimed power draw	236 W	238 W	7 W	35 W	95 W	10 W	65 W
Year of release	2008	2010	2009	2006	2009	2009	2009
Cost at release	£400	£2000	~£60 (integrated)	£3200	£110	£180	£180

GFLOPS of both CPUs are as reported by SIS Sandra.

7.3.3 Problems encountered

Multiple testrigs had to be used due to time and hardware availability constraints, see future work chapter 10.5 for a better solution.

The only AMD GPU hardware that could be acquired was a Radeon 4330 mobility and 4870, Bristol university had two 5870 GPUs on order but they did not arrive in time for this project. Although AMD's OpenCL SDK 2.2 worked well on both AMD and Intel CPUs there were significant issues getting GPUs to work with it. The 4870 simply crashes the entire system if BUDE is made to run on it, although the lines of code that were causing this were eventually located they were crucial to BUDE's operation and were valid OpenCL. The 4330 mobility also suffered numerous issues, it would only work if Catalyst 10.4 was used and only worked with SDK2.1. Also the clGetDeviceInfo OpenCL call reports the maximum work group size for the GPU is 128, however after extensive trial and error testing it was determined the true maximum work group size was 32. This is a bug in either the driver or the SDK. AMD's Radeon 4000 series

of GPUs only have beta OpenCL support so it is possible a Radeon 5000 series would run BUDE correctly but, as mentioned earlier, none were available for testing. This is further supported by other OpenCL programs failing to run on Radeon 4000 series but working on 5000 series [24].

8 Results

Figure 8 shows the performance results of the test along with the initial cost of each card. When running on the C2050 'Fermi' it was calculating each iteration of 650,000 poses so fast that the percentage of time spent sorting energies on the host was becoming a significant amount of time relative to time spent calculating energies on the card. Performance has increased to such an extent that BUDE is now coming up against Amdahl's law.

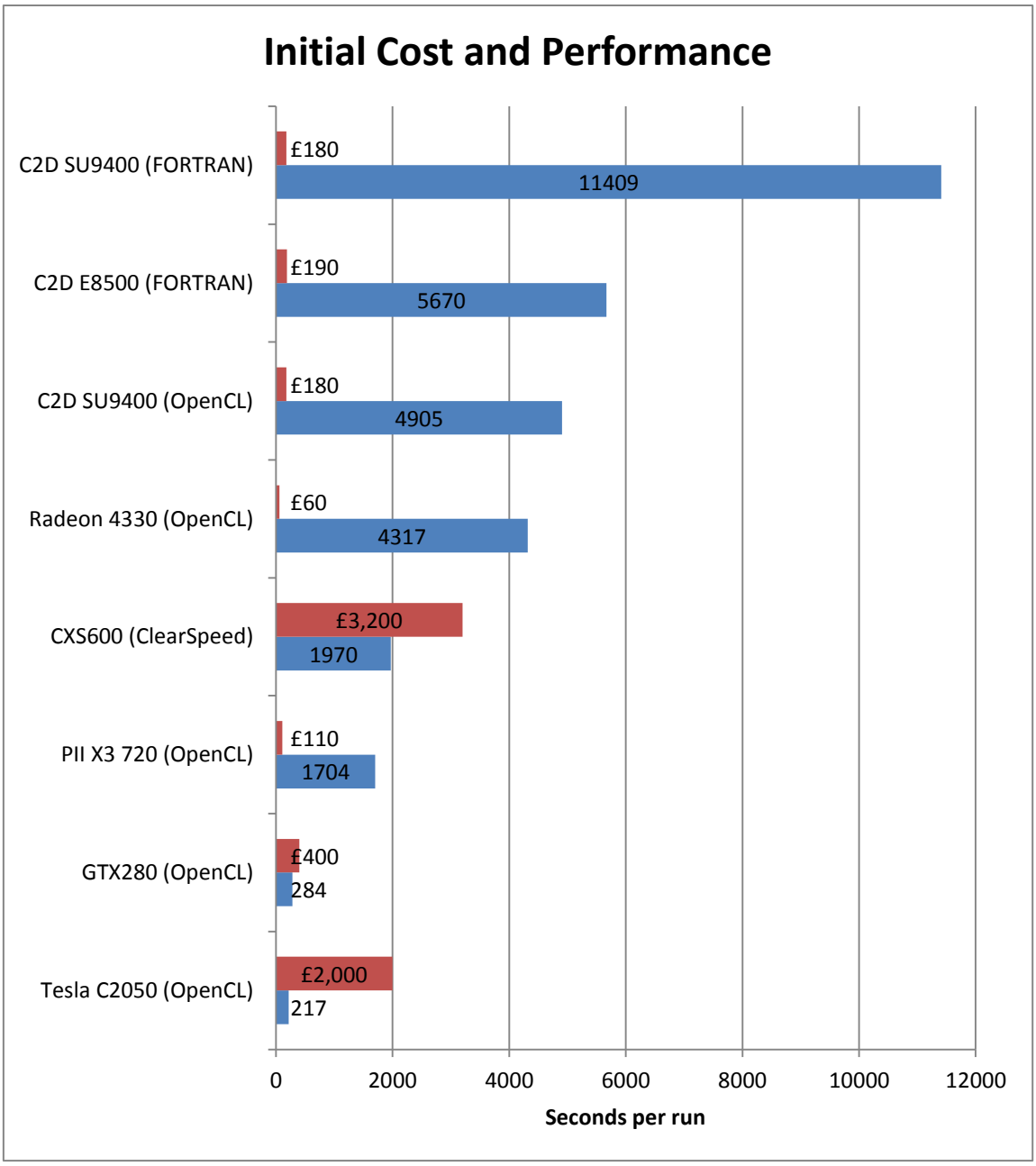

Figure 8

OpenCL acceleration gives a significant performance boost. The fastest card is the Nvidia Tesla C2050 'Fermi' however it only beats the Nvidia GTX280 by a small amount, 217 to 284 seconds respectively. The Radeon 4330's performance is handicapped because its theoretical performance figure of 72 GFLOPS presumes the code makes full use of 4 way vectorisation. As the OpenCL implementation of BUDE contains no vectorisation this would significantly reduce the utilisation of the GPU and hence its performance.

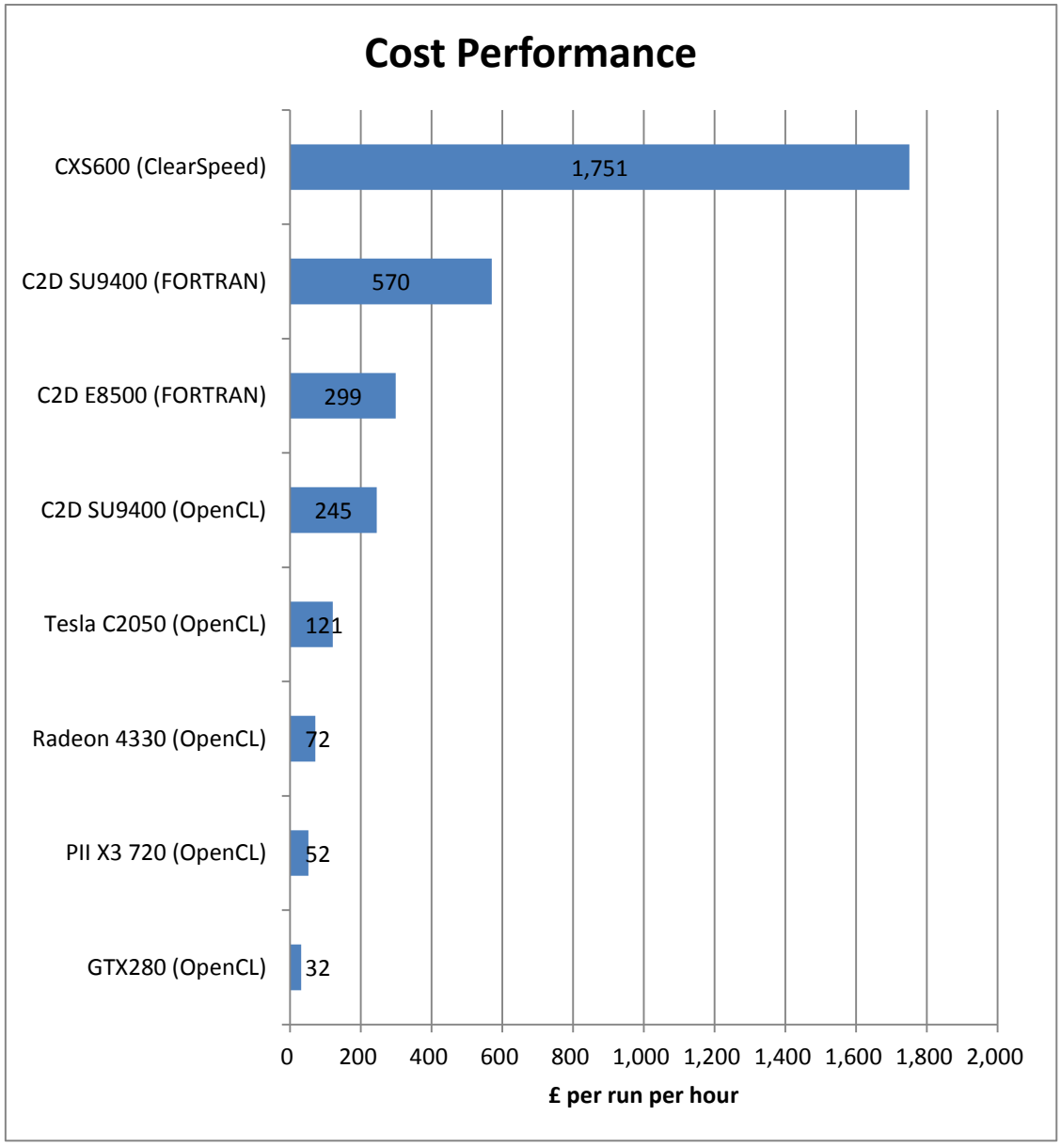

Figure 9

Figure 9 shows that Nvidia's GTX280 provides the best performance for the initial price. This would be different if the consumer version of 'Fermi', the GTX480, had been used. It is likely 'Fermi' would be a lot more competitive with the GTX280 if this was the case.

Figure 10 shows the arithmetic mean average system power consumption over an entire run on an experiment using BUDE. These results should be taken with caution because they were not done on an 'apples to apples' basis. The test rigs had very significant variations in idle power draw, from 11 watts on test rig 5 to 159 watts on test rig 2. However a general trend is visible that shows GPUs are more efficient in energy used per calculation than an equivalent CPU. If the ClearSpeed CSX600 was in a system with very low idle such as the test rig 5 it would be expect to use as little power as the Radeon 4330 or possibly even less.

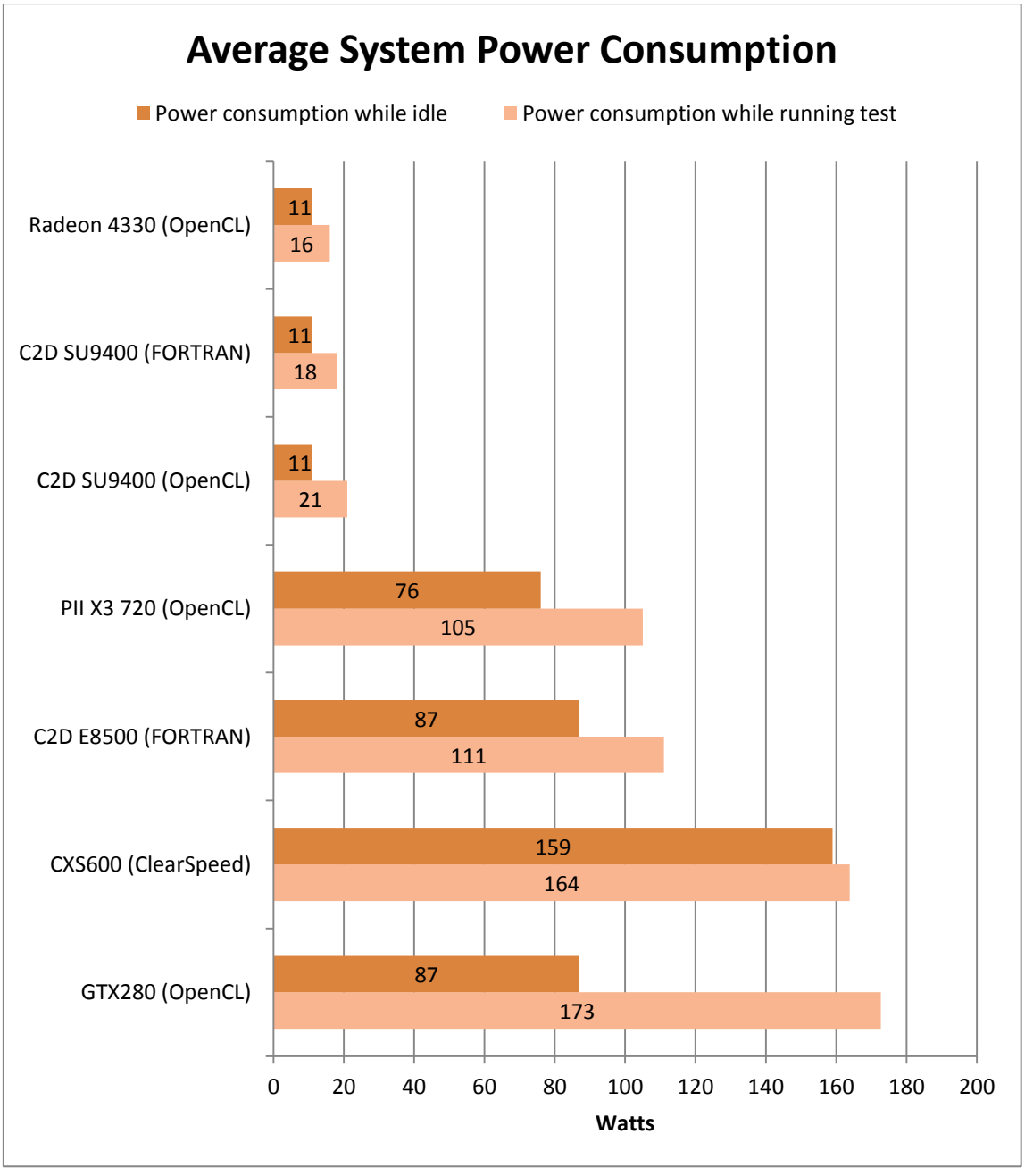

Figure 10

Figure 11 shows the power profile of the ClearSpeed CSX600 and Nvidia GTX280 accelerators taken as an average over five runs. An important point to remember when looking at these

results is the idle power consumption of the testrigs being used. The test rig the GTX280 ran on had an idle power draw of 87 watts while the test rig the CSX600 was running in had an idle of 159 watts.

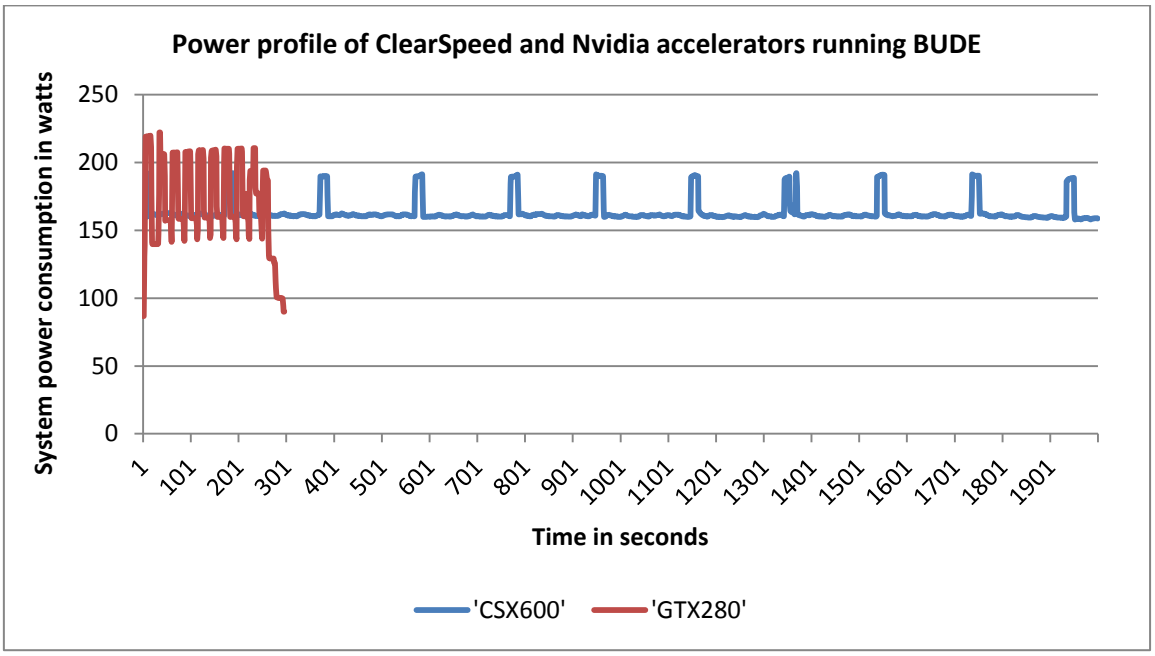

Figure 11

To determine the cause of the peaks and troughs in the power profile, print statements were inserted into the code to identify what stage the program was at and the power usage was manually observed in real time. The peaks visible in the GTX280's power profile corresponded to when the GPU was under full load calculating energies, the troughs are where the FORTRAN code is sorting energies and preparing the next iteration. The CSX600's power profile is the complete opposite of the GTX280. The peaks correspond to the FORTRAN code sorting energies and the troughs are where the CSX600 is calculating energies. The ClearSpeed CSX600 card uses so little energy that the system is consuming just 2 watts more under load than when idle. This small power draw is because the ClearSpeed version of BUDE is unable to use the CSX600 at full speed further resulting in reduced power draw.

Figure 12 shows the power used per experiment run in kWh. The laptop CPU and GPU are of particular interest as they are both on the same platform and are both low power parts so should be a fair comparison.

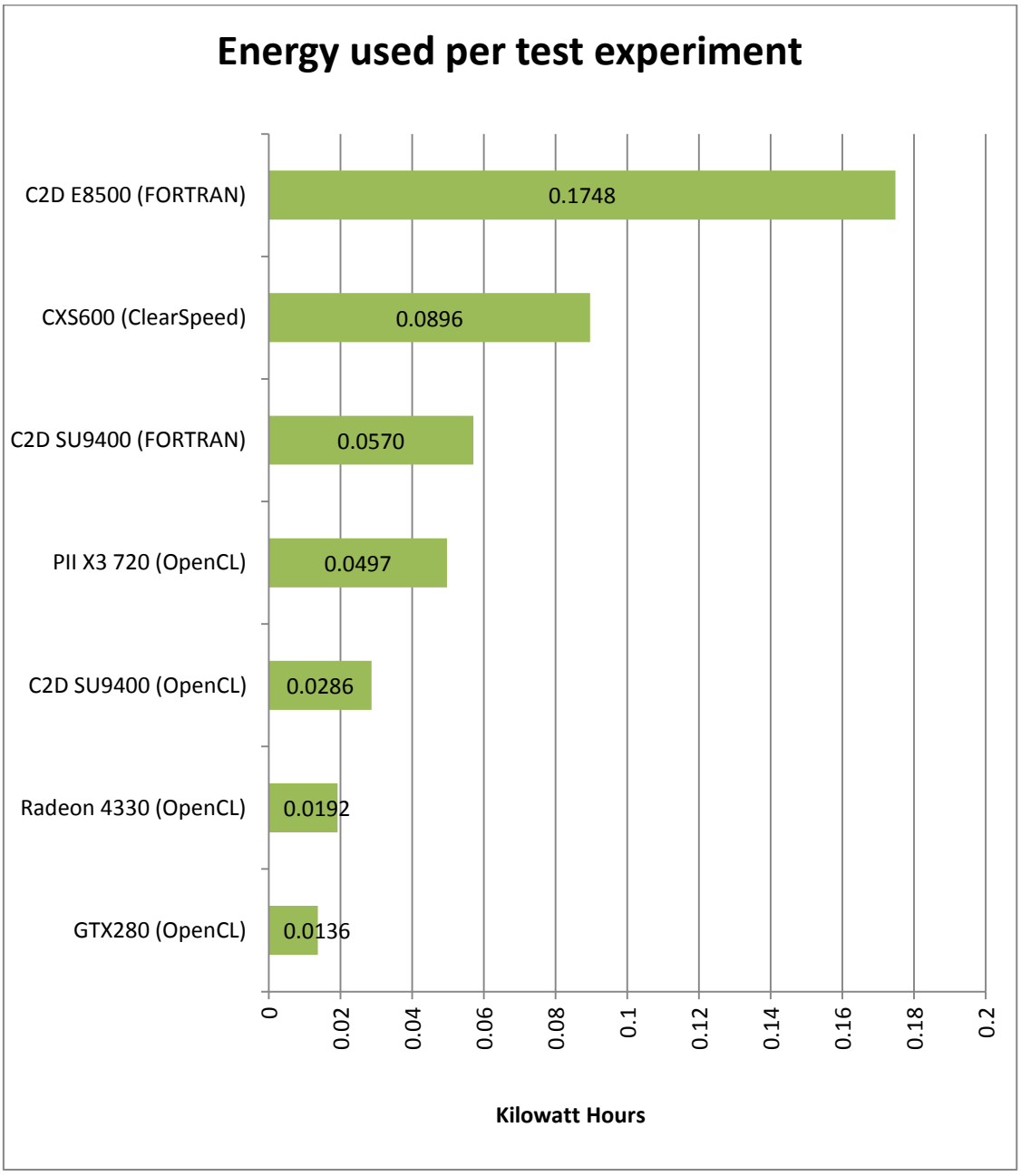

Figure 12

Table 5, energy data table

Computation device	kWh used per run	Cost per run in £GBP	C02 emitted per run in grammes
GTX280 (OpenCL)	0.014	£0.00191	7.22
Radeon 4330 (OpenCL)	0.019	£0.00269	10.17
C2D SU9400 (OpenCL)	0.029	£0.00401	15.16
PII X3 720 (OpenCL)	0.050	£0.00696	26.34
C2D SU9400 (FORTRAN)	0.057	£0.00799	30.23
CXS600 (ClearSpeed)	0.090	£0.01255	47.51
C2D E8500 (FORTRAN)	0.175	£0.02448	92.66

Table 5 shows the kWh used per run as displayed in figure 12 alongside the cost per run and C02 emitted. Cost per run is based on 1 kWh of electricity costing £0.14, this was chosen as it is a typical domestic tariff. The figures used for C02 emitted per run are based on the assumption that 530 grammes of C02 are emitted per kWh of energy used; this assumes the system is being powered off the UK National Grid. The 530g figure is an average obtained from the UK Grid Carbon Intensity application, 'GridCarbon', available on Apple's iOS operating system.

Although the Nvidia GTX280 is clearly the cheapest card none of them are expensive to run for a single experiment. To accurately assess the electricity cost for running BUDE it would be necessary to know how many hours it would be run for per year. Assuming the most extreme case, where BUDE is run 24/7, 356 days per year, then the electricity cost for using the GTX280 system would be £212.17, but during this time it would have performed 111,042 simulation runs. Compare this to an unaccelerated system such as the dual-core E8500 which would cost £136.13 in electricity for the year but only perform 5,562 simulation runs.

Concerns about the effects of man-made C02 emissions on Earth's climate make the C02 emitted per experiment an important measure. The GTX280 is, once again, the most efficient device tested. If the code was significantly vectorised it is likely the Radeon 4330 would match the GTX280's efficiency.

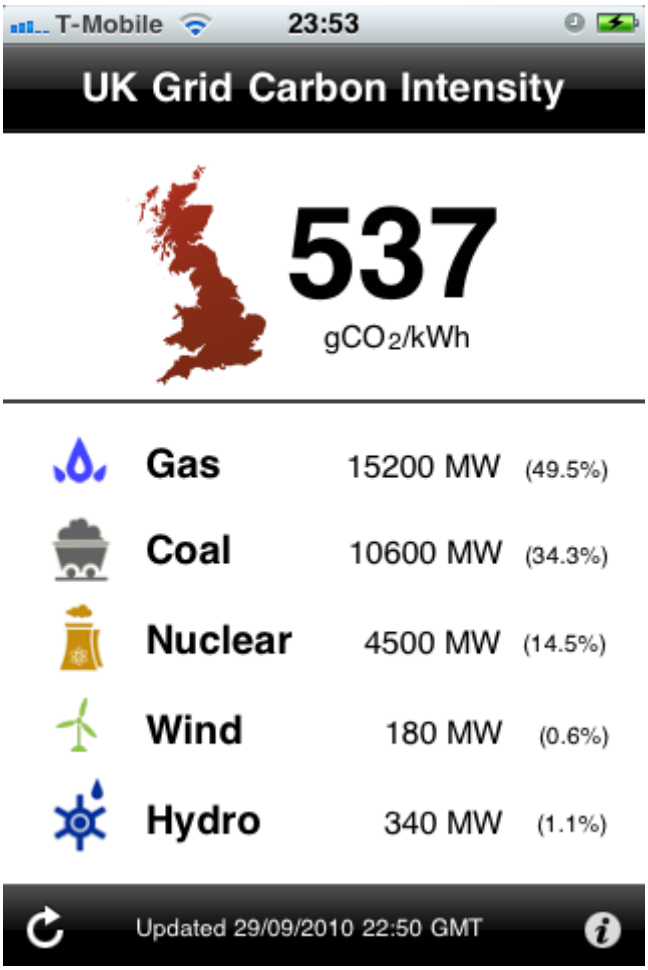

Figure 13, screenshot of UK Grid Carbon Intensity app

Figure 13 shows a screenshot from the UK Grid Carbon Intensity application displaying the instantaneous carbon intensity at 22:50 GMT on 29/09/2010 and the makeup of the Grid's electricity sources at this time. The average UK grid carbon intensity is approximately 530g, this screenshot was taken at a time where the figure is above average. This figure is expected to significantly reduce in the future as the makeup of electricity sources move away from C02 emitting fossil fuels to carbon free sources such as nuclear and renewables.

9 Conclusions

The four objectives of this project were to:

1) Implement a version of BUDE in OpenCL that works on x86 CPUs.

This objective has been fully met. The new OpenCL implementation of BUDE has worked correctly on every CPU tested and with significant performance improvements over the original FORTRAN version of BUDE.

2) Make necessary modifications to allow the OpenCL kernel to run on GPUs.

This objective has been met but it failed to run on one of the GPUs tested, the Radeon 4870. The reason for the failure of BUDE to run on the 4870 was unable to be determined in time but it is unlikely to be due to a bug in BUDE's code. AMD's Radeon 4000 series only has beta support for OpenCL and BUDE successfully ran on a Radeon 4330, albeit with a modification to the code due to an OpenCL API call returning an incorrect figure.

3) Optimise the program to improve performance.

Optimisations were carried out that resulted in performance improvements of approximately 10-20%. The improvement in performance due to optimisation was biggest on the fastest GPUs such as the GTX280 and smallest on CPUs where it made no discernable difference. The performance improvements mainly focussed on improving how many work items were queued for execution and how many memory accesses were performed per kernel both of which benefit GPU architecture more than CPU.

4) Benchmark the program on x86 CPU, GPU and ClearSpeed accelerator platforms then determine the most efficient compute device.

Benchmarks were successfully performed on 8 systems with power readings taken on 7 of those. The results show a performance improvement of 9x when using a C2050 'Fermi compared to the CXS600 currently being used. Due to hardware availability it was not possible to compare a directly equivalent CPU and GPU but an Nvidia GTX280 runs BUDE 6x faster than an AMD Phenom II X3 720. It was also determined the GTX280 is the most efficient device of those tested for running BUDE in terms of performance per watt, performance per initial price, kWh per experiment and C02 emissions per experiment.

The speedup provided using a Tesla C2050 or a GTX280 will not only be more efficient in energy terms but will make BUDE more useable due to the speed at which results will be available. It will be possible to perform more experiments with more drugs and should help identify possible drug candidates faster than before.

Although the GTX280 is the most efficient card for a single experiment, it does not necessarily follow that it would use less electricity and emit less C02 over its lifetime. This would only be true if the same number of BUDE experiments were run per year regardless of how fast they are performed. As Figure 10 shows it has the highest average power consumption of any system tested. In reality the higher performance will make BUDE more usable and is therefore likely to be used for at least the same number of hours per year if not more. This would result in it using more electricity and emitting more C02 over its lifetime. To offset this BUDE could be run at off peak times to minimise the C02 emitted per kWh of energy used.

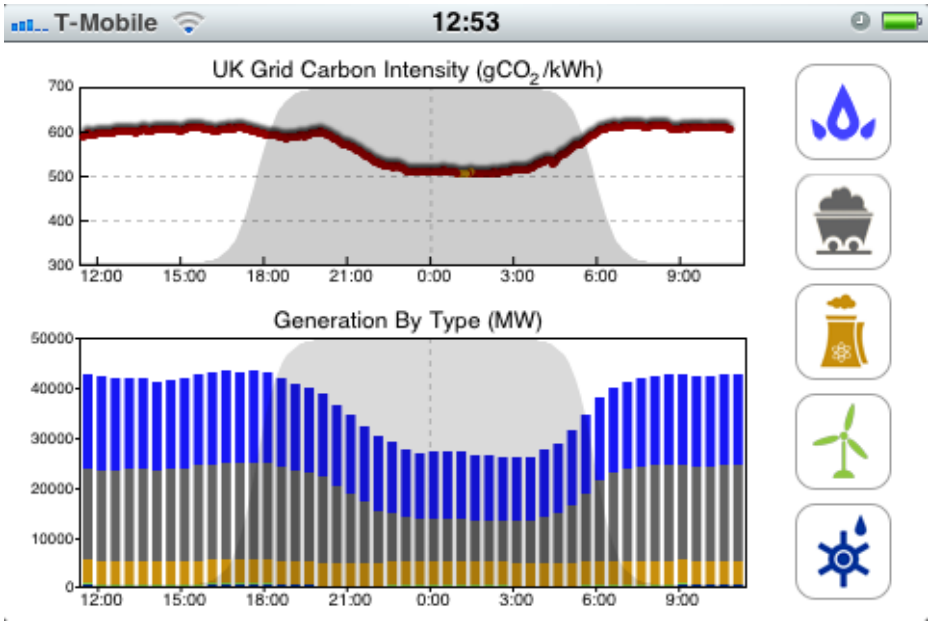

Figure 14, screenshot of UK Grid Carbon Intensity app

Figure 14 shows the daily fluctuation in carbon intensity and specifically that it is lowest overnight between 23:00 and 3:00. It would be possible to lower carbon emissions by running experiments at this time. If electricity is purchased through industrial contracts it is possible to exploit the night/day energy demand curve [26] and get cheaper electricity at off peak times (approximately between 21:00 and 5:00).

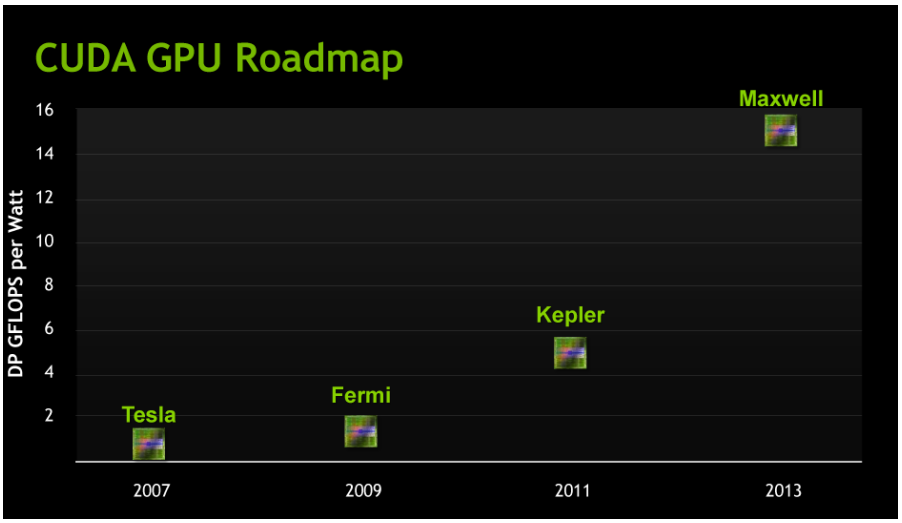

Figure 15, screenshot of Nvidias roadmap for future GPUs announced at GPU Technology Conference 2010

Nvidia claims future GPUs will not only have higher performance but will also significantly improve in terms of performance per watt. Figure 15 shows how double precision GFLOPS per watt is expected to improve exponentially in future GPU hardware, the OpenCL implementation of BUDE created for this project will be capable of exploiting this.

10 Future work

Although this project has achieved significant performance improvements there is still room for extra optimisation to further increase performance. The OpenCL kernel code is already well optimised in regards to minimising the number of conditional statements and the size of loops so any time spent looking for further optimisations here is likely to result in minimal improvements. The following areas have been identified that do have a significant potential to further increase the performance of BUDE.

10.1 Optimise kernel for specific hardware

The two most important OpenCL vendors, ATI/AMD and Nvidia, provide 'performance profilers' with their OpenCL SDKs. These tools record various statistics about a program while it runs on a GPU giving the programmer and indication of where bottlenecks might be. The bottlenecks could be in different places on different GPU architectures so it is best to find out from the GPU vendor what kind of values you should be hoping to achieve on the GPU being used for profiling.

This project has been profiled using Nvidia's profiler on a GTX280 platform. It was only profiled on this single platform due to limited time and AMD's OpenCL implementation having issues with BUDE. Table 6 shows a cut down version of the results from the profiler.

Table 6, results from Nvidia's OpenCL profiler running BUDE on a GTX280

%Time on GPU	99.91
%Time copying from host to GPU	0.08
%Time copying from GPU to host	<0.005
Global mem overall throughput (GB/s)	44.9756
Occupancy (GPU usage)	0.5
Average branch:divergent branch ratio	300:1
Local loads	30,290,600
Local stores	30,392,300
Global 32 bit loads	13,557,900
Global 128 bit loads	11,520
Global 64 bit stores	338,233,000

The profiler shows that the kernel is already running efficiently but there are some areas for improvement. The first three statistics show that the kernel is spending 99.91% of its time running on the GPU as opposed to transferring data between the host and GPU and vice versa. The next statistic shows that the Global memory is being accessed at a rate of 44.96GB/s which is well below the GTX280's maximum bandwidth of 141.7GB/s so there is no need to shrink this

as BUDE is known to be compute limited rather than memory bandwidth limited. Occupancy is the most important statistic. It tells you how many of the GPU's cores are being utilised and therefore how efficiently the GPU is being used. The theoretical maximum is 1 which would indicate all cores are being used all the time, but for GPGPU purposes the realistic maximum that could be attained according to Nvidia is between 0.6 and 0.7. An occupancy of 0.5 is therefore already very good, particularly considering the next statistic, the number of divergent branches relative to non-divergent branches. A divergent branch is where one core branches in one direction and another in a different direction so they become out of sync in terms of what stage they are at in the program's execution. In the GTX280 cores (known as stream processors) are grouped together into 'streaming multiprocessors', or SMs, in which they share resources such as a single instruction fetch unit and a single program counter. Because of these shared resources divergent branching can significantly impact on the performance. A divergent branch ratio of 300:1 is quite good considering the significant amount of branching that happens in the code and, as the occupancy shows, it does not seem to be significantly affecting the performance. The final five statistics are showing the total number of local and global stores/loads. The most important ones to look at are the global reads and writes to the graphics card's DRAM as these are several orders of magnitude slower than local accesses to on-chip memory. Global stores/loads can be either 32, 64 or 128 bits wide, each of which take the same amount of time to perform because the GPU's memory interface is 128-bit wide. This means doing 4 separate 32 bit reads takes 4 times longer than a single 128 bit read so ideally as many stores/loads as possible will be 128 bit. As the table shows this is not the case with BUDE's OpenCL kernel so there is strong potential for further speed increases by modifying the code to achieve this. The most obvious method to minimise the number of 32 bit stores/loads is to make the kernel process the energies for four protein atoms per iteration instead of one. This would allow the relevant loads and stores to be coalesced together, shortening the time spent stalled waiting for a variable to arrive from global memory and potentially pushing up the occupancy from 0.5.

The memory performance could also be improved by making use of the constant and local memories. These two on-chip memories are shared between all processing elements like global but are at minimum an order of magnitude faster than global memory. It is unlikely that the entirety of all the global arrays could fit into the constant and local memory of current generation GPUs but loops that use them could be broken down into smaller chunks allowing parts of the global arrays to move to faster memory. By making use of OpenCL's API calls it is possible to query the hardware, find the size of its constant and local memory then dynamically allocate parts of arrays to it by passing the memory sizes through to the kernel.

Another way to optimise the kernel for specific platforms would be to include vendor specific code or OpenCL extensions. OpenCL is a standardised programming model designed to allow programs to run without modification on any platform that supports the OpenCL API. To enable this flexibility performance is sacrificed when compared to platform specific parallel programming languages such as CUDA on Nvidia cards. However because OpenCL is designed specifically for high performance programs it has one of two options if you wish to tailor a program to a specific vendor or piece of hardware and break the cross platform compatibility.

The first and simplest option is to make use of performance enhancing OpenCL extensions provided in certain vendors OpenCL implementations. The second option is to insert higher performance non OpenCL code directly into the OpenCL kernel at key points, such as Nvidia CUDA.

10.2 Vectorising the kernel

Another potential area for improvement is vectorising the kernel. Vector calculations are supported in the OpenCL standard and can provide significant speed increases on SIMD hardware such as CPUs with SIMD extensions or any of AMD's OpenCL compatible GPUs. Individual Nvidia GPU cores are scalar and lack any SIMD capability so Vectorisation should have no effect on the arithmetic performance, load/store performance however could be improved as previously mentioned. Two areas of the kernel have already been identified as potential targets for vectorisation at lines 130 to 135 and 98 to 100. Unrolling the protein atom loop in fasten3() four times would quadruple the number of atoms processed per iteration, and thus as described in chapter 10.1 would provide further potential for Vectorisation. This would be particularly beneficial for AMD GPU's which require the use of four element vectors to achieve maximum performance. It would also be beneficial for running OpenCL on host CPUs where Vectorisation would enable the exploitation of the x86 architecture's SIMD instruction sets such as SSE and AVX.

10.3 Split workload

The kernel isn't the only part of code that has potential for optimisation. The OpenCL host code currently only supports a single OpenCL device at any one time. By making use of various OpenCL API calls such as clGetPlatformIDs and clGetDeviceIDs it is possible for the host to make use of all available OpenCL devices simultaneously. The biggest potential for speed increase by changing this would be if a computer had 2 or more GPUs but even if the computer has just one GPU it would be possible to get a ~10% performance increase by making use of the CPU which is normally idle while an OpenCL kernel is running (only currently possible if using the AMD OpenCL implementation). 2-4 GPU server blades and workstations are common for GPGPU supercomputers, and hardware vendors are now designing chassis to hold very dense installations of GPUs, such as Dell's new 3U PowerEdge C410x 16 GPU expansion chassis. Hence supporting multiple GPUs is a vital change that should be made to BUDE to allow for scalability. Presuming the workload (number of transforms/poses) is large enough this should allow the performance to scale in a near linear fashion with each extra GPU added.

We should note that, if running relatively small numbers of simultaneous poses (less than 50,000 per EMC iteration) it would be faster to simply run multiple instances of the entire BUDE program. This approach was supported in the original ClearSpeed version of BUDE and still works in the new OpenCL version.

10.4 MPI

BUDE will be running on a new part of Bristol University's Blue Crystal supercomputer which will contain 8 Nvidia Tesla M2050 GPUs spread across 4 1U servers. To make full use of this processing power would require 8 instances of the current OpenCL version of BUDE to be run simultaneously. If small numbers of poses are being processed per EMC iteration this would be the optimal way to run BUDE. However if the user desired to get the results of a single instance of BUDE that is doing a large number of poses (1 million or more) as fast as possible there are improvements that can be made to make this possible. If the changes specified in chapter 10.3 were made a single instance of BUDE could use all the available GPUs on a single server, which would be 2 in the case of the new GPU nodes in Blue Crystal. This is an improvement but still limits a single instance of BUDE to running on one physical computer. By using MPI it is possible to have one instance of BUDE use compute resources from multiple computers networked together such as in Blue Crystal.

BUDE already has an MPI implementation. The current implementation is at the frontend and automatically creates multiple instances of BUDE on multiple systems then distributes the necessary data to ensure the separate instances do not perform duplicate calculations. This would be best suited for situations with less than 50,000 poses. MPI could also be inserted in a different part of the program to speed up a single calculation by distributing it across multiple systems. This would involve inserting the MPI code deeper in BUDE, effectively splitting the host code into two as shown in Figure 16. The MPI would be inserted into the calgen() routine inside BUDE which would allow it to distribute energy calculation across multiple systems.

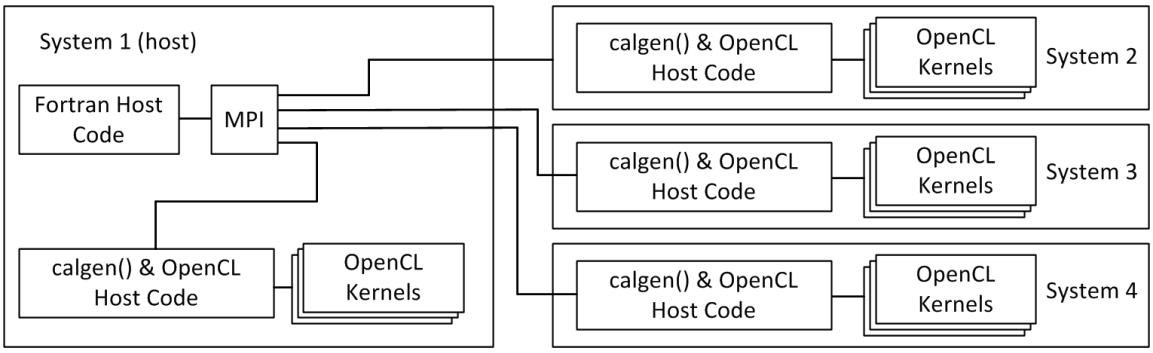

Figure 16, possible configuration of BUDE with MPI

As described earlier this would allow a large number of poses to be calculated quickly. The number of poses would have to be sufficiently high that the added overheads of MPI are not the dominant factor determining performance.

10.5 Optimise host code

The final piece of recommended future work is implementing the entirety of BUDE's host code in a modern multi-threadable programming language such as C or C++ instead of FORTRAN.

Originally, in the FORTRAN only version of BUDE, energy calculations took over 99% of the compute time. GPU acceleration increases the speed of this part by such a significant amount that the host code is becoming the limiting factor. Unfortunately most of the host code is unsuitable for OpenCL GPU acceleration due to a lack of data parallel workloads but it should benefit from task parallelism however a couple of small areas have been identified that could benefit from OpenCL acceleration. These two areas are the conversion of the "pose descriptors" into transformation matrices and the sorting of the energy calculation results to get the best energy structures.

Increasing the performance of the rest of BUDE's host code would best be achieved by implementing it in C or C++ and making use of threads. It could be possible to implement more of the host in OpenCL and run it solely on CPU OpenCL devices but unless large parts could be vectorised to make use of SSE there would be no performance gain over multithreaded C/C++ leaving only the disadvantage of unnecessary complexity and OpenCL overheads.

Another, easier, way of increasing the speed of the FORTRAN host code would be to use a better optimised compiler than f77 such as Intel's FORTRAN compiler. To make Intel's compiler work various changes would have to be made to the code but it should result in a one off boost of ~50%.

Work has already begun on translating the main bulk of BUDE from FORTRAN77 to C++ separately from this project.

10.5 Further benchmarking

Not only is there further work to do in optimising the code, there are many more benchmarks that could be done.

Due to time and hardware availability limitations many of the benchmarks in this project were not done on an apples to apples basis. This means that the Nvidia GTX280 was benchmarked in a system that idles at 87 watts against a ClearSpeed CXS600 in a system that idles at 160 watts. Further benchmarks could be done that compare the two cards in more fair circumstance. This would involve using a single system and simply swap cards so all other things are equal. To ensure there are no software driver clashes from previously installed accelerators it would be best to format the system after each card has been tested five times. This is particularly important if benchmarking an AMD GPU then an Nvidia GPU and vice versa.

As well as performing the benchmarks under better conditions there are extra pieces of hardware that should be benchmarked. This includes an Intel i7 CPU to test a top of the range CPU and an ARM Cortex A8 or A9 to test if energy saving CPUs can compete with high performance CPU in terms of performance per watt. Similarly Nvidia's 'Fermi' could be compared to a power saving mobile CPU such as a Power VR SGX series GPU. Most importantly however would be benchmarking AMD's GPUs such as the 5870 'cypress' and future AMD products such as the upcoming 'southern islands' 6870 and Fusion APU as these are direct

competition for the Nvidia ‘Fermis’ BUDE will be running on . It would also be possible to benchmark BUDE on a ClearSpeed CATS box which contains 12 ClearSpeed CXS600 cards.

11 Bibliography

[1] S.Smith. *Cyclic peptides as protease inhibitors: development of a combinatorial library approach guided by computational modelling*. (2005). PhD thesis, University of Bristol.

[2] F. Liang and W. H. Wong. **Evolutionary Monte Carlo for protein folding simulations**. *Journal of Chemical Physics*, 115(7):3374--3380, 2001.

[3] Richard Sessions, Simon McIntosh-Smith, **An Accelerated, Computer Assisted Molecular Modeling Method for Drug Design (2008)**, *International SuperComputing, Dresden*

[4] http://forum.pcvsconsole.com/viewthread.php?tid=12946 [May 2010] and http://support.clearspeed.com/documentation/hardware/csx600/ [May 2010]

[5] http://en.wikipedia.org/wiki/File:Processor_families_in_TOP500_supercomputers.svg [April 2010]

[6] John E. Stone, Jan Saam, David J. Hardy, Kirby L. Vandivort, Wen-mei W. Hwu, and Klaus Schulten. **High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs (2009)**. *In Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units*, *ACM International Conference Proceeding Series*, volume 383, pp. 9-18,

[7] David E. Shaw, Martin M. Deneroff, Ron O. Dror, Jeffrey S. Kuskin, Richard H. Larson, John K. Salmon, Cliff Young, Brannon Batson, Kevin J. Bowers, Jack C. Chao, Michael P. Eastwood, Joseph Gagliardo, J.P. Grossman, C. Richard Ho, Douglas J. Ierardi, István Kolossváry, John L. Klepeis, Timothy Layman, Christine McLeavey, Mark A. Moraes, Rolf Mueller, Edward C. Priest, Yibing Shan, Jochen Spengler, Michael Theobald, Brian Towles, and Stanley C. Wang . **Anton, A Special-Purpose Machine for Molecular Dynamics Simulation (2008)**. *Communications of the ACM* (ACM) 51 (7): 91–97. ISBN: 978-1-59593-706-3.

[8]David E. Shaw, Ron O. Dror, John K. Salmon, J. P. Grossman, Kenneth M. Mackenzie, Joseph A. Bank, Cliff Young, Martin M. Deneroff, Brannon Batson, Kevin J. Bowers , Michael P. Eastwood, Douglas J. Ierardi, John L. Klepeis, Jeffrey S. Kuskin, Richard H. Larson, Kresten Lindorff-Larsen ,Paul Maragakis, Mark A. Moraes, Stefano Piana, Yibing Shan, Brian Towles, **Millisecond-scale molecular dynamics simulations on Anton (2009),** *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* ISBN:978-1-60558-744-8

[9] http://en.wikipedia.org/wiki/Anton_(computer) [April 2010]

[10] James C. Phillips, John E. Stone , **Probing Biomolecular Machines with Graphics Processors (2009)**, *Communications of the ACM* 52(10):34-41, 2009.

[11] Stone, J.E., Phillips, J.C., Freddolino, P.L., Hardy, D.J., Trabuco, L.G., and Schulten, K. **Accelerating molecular modeling applications with graphics processors (2007).** *Journal of Computational Chemistry*, ISBN:28-2618-2640.

[12] http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html [May 2010]

[13] http://fhtr.blogspot.com/2010/03/gazing-at-cpu-gpu-crystal-ball.html [May 2010]

[14] C. Angelini, F. Abi-Chahla. **Double Or Nothing.** http://www.tomshardware.com/reviews/radeon-hd-5870,2422-3.html [May 2010]

[15] R.Smith, **Nvidia's GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?** http://www.anandtech.com/show/2977/nvidia-s-geforce-gtx-480-and-gtx-470-6-months-late-was-it-worth-the-wait-/6 [May 2010]

[16] http://forums.nvidia.com/index.php?showtopic=81606 [May 2010]

[17] David B. Kirk, Wen-mei W. Hwu. **Programming Massively Parallel Processors, A Hands-on Approach (2009)**. p 2-7, ISBN: 978-0-12-381472-2.

[18] Timothy Mattson, Beverly Sanders, Berna Massingill. **Patterns for Parallel Programming (2004).** ISBN13: 9780321228116

[19] David B. Kirk, Wen-mei W. Hwu. **Programming Massively Parallel Processors, A Hands-on Approach (2009).** p 13-15, ISBN: 978-0-12-381472-2.

[20] Khronos group http://www.khronos.org/ [May 2010]

[21] Erich Elsen, Mike Houston, V. Vishal, Eric Darve, Pat Hanrahan, Vijay Pande. **N-Body simulation on GPUs (2006).** *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, article:188, ISBN:0-7695-2700-0

[22] http://ati.amd.com/technology/streamcomputing/opencl.html [May 2010]

[23] http://www.top500.org/ [September 2010]

[24] http://www.phoronix.com/scan.php?page=article&item=opencl_nvidia_ati&num=3 [September 2010]

[25] http://arstechnica.com/business/news/2010/02/amd-reveals-fusion-cpugpu-to-challege-intel-in-laptops.ars [September 2010]

[26] http://www.nationalgrid.com/uk/Electricity/Data/Realtime/Demand/Demand8.htm [September 2010]

[27] http://blogs.arm.com/multimedia/why-opencl-will-be-on-every-smartphone-in-2014/ [September 2010]

12 Appendices

Only bude_cl_host.cpp, bude_cl_calgen_rot.f and docking_compute.cl are in this appendicies. For Makefile, budle_cl_interface.c, bude_cl_h.h and bude.f see submitted source code.

docking_compute.cl

```
/**
BUDE OpenCL kernel file
**/
#define LIGAND_SUBSET_SIZE 64
#define TRANSFORM_BUFFER_SIZE 960*1000
#define NXB 4

#define     SEM_PROCESS_ATOMS_GO            40
#define     SEM_PROCESS_ATOMS_DONE          41
//#pragma OPENCL EXTENSION cl_amd_printf : enable
//#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable


// The data structure for one atom - 40 bytes
typedef struct _atom{
    float x,y,z;
    float radius,  hphb, hard, nndst, npdst, elsc;
    char  hbtype, name[3];
} Atom;


static inline void fasten3 (int natlig,
                                int natpro,
                                float cutdis,
                                float *transform,
                                int ix, __local Atom *ligand_buffer,
                                __global Atom *ligand_molecule,
                                __global Atom *protein_molecule,
                                __global float* etotals,
                                 float* lx,
                                 float* ly,
                                 float* lz,
                                 float* distij2,
                                 int* ligands_close)
{

          int ilbase,il, ip;                          // ligand and protein index
          Atom ligand_atom;
          Atom protein_atom;


          float etot1, etot2, etot3;      // Total energy - the 3 parts
          float strc_e, dslv_e, chrg_e;   // Components of the tot. energy
          float radij;                    // sum of 2 atom spheres radii
          float distij;                   // distance between 2 atom centres
          float distbb;                   // ball:ball dist (=distij - radij)
          float elcdst,elcdst1;           // halo distance - 4. or 6. and reciprical

          /* temporary variables */
          int lcount;                          // # of ligand atoms on each PE
          int lcount_close;                    // # of ligand atoms on each PE that are <cutdis away from P a
          float distdslv=1.;              // desolvation distance
          float const cnstnt=22.5;        // 22.5 factor
          int  p_action;                 // =1 if one or both charged/bipolar
          int  zone1;                    // which region we are in
          float fact;                     // shape function  0.->1.
          float p_hphb1,l_hphb1,l_hphb;
          int t1;                                 //temp

          const float zero=0., one=1.,half=0.5,four=4.0, six=6.0 ;
          float cutdis2;                          // 10*10


          float x,y,z;                       // temporaries
          /* the properties of one protein atom */
          float p_x, p_y, p_z,               //
                              p_radius, p_hphb, p_hard,   // properties of one Protein
                              p_nndst, p_npdst, p_elsc;   // atom, in __local memory
          float p_elsc_m,p_hphb_m;
          int p_hbtype_m;
          int p_hphb_nzero;                   // if the protein's hphb value is zero

          event_t e, ev;



          // we will use the square of the cutoff distance for speed
          cutdis2 = cutdis * cutdis;

          etot1 = zero; etot2 = zero; etot3 = zero;
```

```
          //
          // Transformation step
          // Here we translate and rotate the ligand atom to its test position
          //
          for (il = 0; il < natlig; il++) {
            /** TODO CHECK THESE USING . INSTEAD OF -> **/

                      x =   ligand_molecule[il].x;  // or use pointers to save local memory?
                      y =   ligand_molecule[il].y;
                      z =   ligand_molecule[il].z;

                      /** TODO  >>>>>vectorise this transformation<<<<<<<< would help SIMD devices **/
                      // Do as 3 loops: x,y,z
                      //   get 4 x values, cs_vecMulacc each with tr[0:3]
                      lx[il] = x*transform[0] + y*transform[1] + z*transform[2]    + transform[3];
                      ly[il] = x*transform[4] + y*transform[5] + z*transform[6]    + transform[7];
                      lz[il] = x*transform[8] + y*transform[9] + z*transform[10]   + transform[11];
          }

          //
          // Loop over all the protein balls
          //
          for (ip=0; ip<natpro;ip++) {

            protein_atom = protein_molecule[ip];

                    // Take a copy of this protein atom into local variables
                    p_x      = - protein_atom.x;  // -ve so we can add - might be quicker?
                    p_y      = - protein_atom.y;
                    p_z      = - protein_atom.z;
                    p_radius = protein_atom.radius;
                    p_hphb   = protein_atom.hphb;
                    p_hphb_m = protein_atom.hphb;
                    p_elsc_m = protein_atom.elsc;
                    p_hbtype_m = protein_atom.hbtype;
                    p_hphb_nzero = (protein_atom.hphb != 0.);
                    p_hard   = protein_atom.hard;
                    p_nndst  = protein_atom.nndst;
                    p_npdst  = protein_atom.npdst;
                    p_elsc   = protein_atom.elsc;

                    //---------Calculate the distance between the ball centres
                    // pre loop over the (16) ligand atoms to compute the set of seperation
                    // distances (squared)

                    /** TODO  >>>>>vectorise me<<<<<<<< would help SIMD devices **/
                    for (il=0; il<natlig; il++) {
                              x = lx[il] + p_x;
                              y = ly[il] + p_y;
                              z = lz[il] + p_z;
                              distij2[il] = x*x + y*y + z*z;
                    }

                    for (il=0; il<natlig;il++) {
                              distij = distij2[il];


                              // reference implimentation of sqrt for a local float
                              distij = sqrt(distij);

                              //------Calculate the sum of the sphere radii
                              radij = p_radius + ligand_molecule[il].radius;
                              distbb = distij - radij;
                              zone1=(distbb < zero);

                              //-----Step 1 : Calculate steric energy for when distance is less than the sum of the
radii

                              if (zone1) {
                                  etot1 += (one - distij/radij)
                                                  //* half*(p_hard + ligand_atom->hard);
                                                  * (p_hard + ligand_molecule[il].hard);

                              }

                              //------Step 2 : Calculate formal and dipole charge interactions.
                              if (p_elsc_m != 0.0) {

                                        //----------First set the constant for the coulombic equn, for formal charge or
dipole interaction
                                        elcdst1=0.5;  elcdst = 2.;
                                        //#ifdef Pinglobal
                                        // note I could do the 'E' test as an elseif ?
                                        if (p_hbtype_m  == 'F') {
                                                  if (ligand_molecule[il].hbtype == 'F') {
                                                            elcdst1 = 0.25; elcdst = 4.;
                                                  }
                                        }

                                        //----------Assign an optimal interaction for non-overlapping spheres, to
spheres within overlapping distance
                                        //----------Then linearly decrease the interaction as the spheres move further
apart

                                        if (distbb < elcdst) { // if an interaction
                                                  //chrg_e = cnstnt * (ligand_atom->elsc * p_elsc);
                                                  // The upper line was uncommented and the line below was commented as
suggested by R. Sessions
                                                  // the commented action was reverted. Crazy results
```

```
                                        chrg_e = ligand_molecule[il].elsc * p_elsc;
                                        if (!zone1) {    // in outer zone so scale back
                                                chrg_e *= (one-distbb*elcdst1);
                                        }

                                        // note that the embedded dataset has no 'E's
                                        // RBS: OR not AND
                                        if (p_hbtype_m == 'E' || ligand_molecule[il].hbtype == 'E') {
                                        // RBS: not < 0 but > 0
                                                if (chrg_e>zero) chrg_e = -chrg_e;
                                        }
                                        etot2 += chrg_e;

                                }  // if an interaction

                        }

                        //-----Step 3 : Calculate the two cases for Nonpolar-Polar repulsive interactions:
                        //-----First is where the ligand is Nonpolar, and the protein is Polar.
                        //-----Followed by a linearly reducing interaction over an additional npdist Angstroms.
                        //
                        // This section doesn't often get invoked as only a value if both atoms are nonzero

                        if (p_hphb_nzero) {                     // mono if
                                //if (protein_atom->hphb != 0.) {
                                p_action=0;
                                l_hphb = ligand_molecule[il].hphb;
                                l_hphb1 = l_hphb;
                                p_hphb1 = p_hphb;
                                if (p_hphb <zero) {
                                        if (l_hphb<zero ) {
                                                distdslv = p_nndst;
                                                p_action=1;
                                        } else if (l_hphb>zero) {
                                                distdslv = ligand_molecule[il].npdst;
                                                p_hphb1 = -p_hphb1;
                                                p_action=1;
                                        }
                                }
                                else {                                  // P must be +ve
                                        if (l_hphb < zero ) {
                                                distdslv =  p_npdst;
                                                l_hphb1 = -l_hphb1;
                                                p_action=1;
                                        }
                                }

                                if (p_action==1) {
                                        if (distbb < distdslv ) {          // if an interaction
                                                //dslv_e = half*(p_hphb1+l_hphb1);
                                                dslv_e = p_hphb1 + l_hphb1;
                                                if (!zone1) {                   // if in outer zone, scale
back
                                                        dslv_e *= (one-distbb/distdslv);
                                                }
                                        etot3 += dslv_e;

                                        }
                                }
                        } // skip if p_hphb is zero
                } // loop ligand atoms
        } // loop over protein molecule


        etotals[ix] = half*etot1 +cnstnt*etot2 + half*etot3;

        return;

}

__kernel void fasten_main(int natlig,
                          int natpro,
                          int ntransforms,
                          float cutdis,
                          __global Atom* protein_molecule,
                          __global Atom* ligand_molecule,
                          __global float* transforms,
                          __global float* etotals,
                          int lbase)
{

  __local Atom   ligand_buffer[1]; /** TODO REMOVE ***/


   float lx[800];   // the xyz of the (16) ligand atoms
   float ly[800];   // handled at a time here
   float lz[800];
   float distij2[800]; // dx^2 + dy^2 + dz^2
   int   ligands_close[800];   // flag the 'to do' list

  float  transform[12];     // n 3x4 transformation matrices per PE

  int i,ix;


  int   testINT;

  /** the 0 stands for workgroup 0 (might change) **/
  ix = lbase + (get_global_id(get_group_id(0)));            // gets the unique work item ID
```

```
  if (ntransforms==-1) return;


  /** copy 12 elements from transforms at a time (12 per transform) **/
  for(i=0;i<12;i++){
    transform[i]=transforms[(ix*12)+i];
  }


  /**Then run fasten **/
  fasten3 (natlig,
             natpro,
             cutdis,
             transform,
             ix,
             ligand_buffer,
             ligand_molecule,
             protein_molecule,
             etotals,
             lx,
             ly,
             lz,
             distij2,
             ligands_close);


}//end of main
```

bude_cl_host.cpp

```
/**
OpenCL host
**/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bude_cl_h.h"

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

//#define MEM_SIZE (128)
#define MAX_SOURCE_SIZE (0x100000)



  cl_device_id device_id = NULL;
  cl_context context = NULL;
  cl_command_queue command_queue = NULL;
  cl_mem memobj_prot_mol = NULL;
  cl_mem memobj_lig_mol = NULL;
  cl_mem memobj_trans = NULL;
  cl_mem memobj_etotals = NULL;
  cl_program program = NULL;
  cl_kernel kernel = NULL;
  cl_platform_id platform_id = NULL;
  cl_uint ret_num_devices;
  cl_uint ret_num_platforms;
  cl_int ret;
  cl_event ev;

  cl_ulong mwg_size;          //max work group size
  cl_int cl_mwg_size=0;

  char errorCode[100];

  FILE *fp;
  char fileName[] = "./docking_compute.cl";
  char *source_str;
  size_t source_size, mwg_size_size;    //max work group size size

  cl_int natlig;
  cl_int natpro;
  cl_int ntransforms;
  cl_float cutdis;

extern "C" {

  int cl_host_startup();
  int cl_create_buffer(int argID, int numBytes);
  int cl_write_to_buff(int argID, int numBytes, void *ptr);
  int cl_write_int(int argID, int numBytes, int ptr);
  int cl_write_float(int argID, int numBytes, float ptr);
  int cl_host_run(int numTransforms, int loop_base);
  int cl_fetch_result(int argID, int numBytes, void *ptr);
  int cl_host_finalize();
```

```
}

int cl_host_startup()
{


  /** Load the source code containing the kernel*/
  fp = fopen(fileName, "r");
  if (!fp) {
    fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -2);
    fprintf(stderr, "Failed to load kernel.\n");
    exit(1);
  }
  source_str = (char*)malloc(MAX_SOURCE_SIZE);
  source_size = fread( source_str, 1, MAX_SOURCE_SIZE, fp);
  fclose( fp );

  /** Get Platform and Device Info */
  ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      fprintf(stderr, "clGetPlatformIDs failed\n");
      exit(1);
  }
  ret = clGetDeviceIDs( platform_id, CL_DEVICE_TYPE_CPU, 1, &device_id, &ret_num_devices);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      fprintf(stderr, "clGetDeviceIDs failed\n");
      exit(1);
  }

  /** Create OpenCL context */
  context = clCreateContext( NULL, 1, &device_id, NULL, NULL, &ret);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      fprintf(stderr, "clCreateContext failed\n");
      exit(1);
  }

  /** Create Command Queue */
  command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      fprintf(stderr, "clCreateCommandQueue failed\n");
      exit(1);
  }


  clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE, sizeof(mwg_size), &mwg_size, &mwg_size_size);
  printf("CL_DEVICE_MAX_WORK_GROUP_SIZE = %d\n", (int)mwg_size);

  cl_mwg_size = mwg_size;

  return 0;

}

/** Create buffer. Only necessary for non-scalars

int argID: kernel argument number
int numBytes: buffer size in bytes

**/
int cl_create_buffer(int argID, int numBytes){

  /** Create buffer object for non scalar args */
  switch (argID) {
    case 0:
      //no need to create buffer for scalar
      ret = CL_SUCCESS;
      break;
    case 1:
      //no need to create buffer for scalar
      ret = CL_SUCCESS;
      break;
    case 2:
      //no need to create buffer for scalar
      ret = CL_SUCCESS;
      break;
    case 3:
      //no need to create buffer for scalar
      ret = CL_SUCCESS;
      break;
    case 4:
      memobj_prot_mol = clCreateBuffer(context, CL_MEM_READ_WRITE, numBytes, NULL, &ret);
      break;
    case 5:
      memobj_lig_mol = clCreateBuffer(context, CL_MEM_READ_WRITE, numBytes, NULL, &ret);
      break;
    case 6:
      memobj_trans = clCreateBuffer(context, CL_MEM_READ_WRITE, numBytes, NULL, &ret);
      break;
    case 7:
      memobj_etotals = clCreateBuffer(context, CL_MEM_READ_WRITE, numBytes, NULL, &ret);
```

```
      break;
  }
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ );
      fprintf(stderr, "clCreateBuffer failed\n");
      exit(1);
  }

  return 0;
}

/** Store non-scalar arg

int argID: kernel argument number
int numBytes: non-scalar size in bytes
void *ptr: pointer to non-scalar

**/
int cl_write_to_buff(int argID, int numBytes, void *ptr){

  switch (argID) {
    case 4:
      ret = clEnqueueWriteBuffer(command_queue, memobj_prot_mol, CL_TRUE, 0, numBytes, ptr, 0, NULL, NULL);
      break;
    case 5:
      ret = clEnqueueWriteBuffer(command_queue, memobj_lig_mol, CL_TRUE, 0, numBytes, ptr, 0, NULL, NULL);
      break;
    case 6:
      ret = clEnqueueWriteBuffer(command_queue, memobj_trans, CL_TRUE, 0, numBytes, ptr, 0, NULL, NULL);
      break;
    case 7:
      ret = clEnqueueWriteBuffer(command_queue, memobj_etotals, CL_TRUE, 0, numBytes, ptr, 0, NULL, NULL);
      break;
    default:
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ );
      fprintf(stderr, "cl_write_to_buff argID not recognised!\n");
  }
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ );
      fprintf(stderr, "clEnqueueWriteBuffer failed\n");
      exit(1);
  }

  return 0;
}

/** Store integer arg

int argID: kernel argument number
int numBytes: int size in bytes
int ptr: integer value

**/
int cl_write_int(int argID, int numBytes, int ptr){

  switch (argID) {
    case 0:
      natlig = ptr;
      break;
    case 1:
      natpro = ptr;
      break;
    case 2:
      ntransforms = ptr;
      break;
    default:
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ );
      fprintf(stderr, "cl_write_int argID:%d not recognised!\n", argID);
  }
  return 0;
}

/** Store float arg

int argID: kernel argument number
int numBytes: float size in bytes
float ptr: float value

**/
int cl_write_float(int argID, int numBytes, float ptr){

  switch (argID) {
    case 3:
      cutdis = ptr;
      break;
    default:
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ );
      fprintf(stderr, "cl_write_float argID:%d not recognised!\n", argID);
  }
  return 0;
}

/** Prints out errfor for setKernelArg calls **/
int setKernelArgError (int retError, int lineNum){

  fprintf(stderr, "Error in %s @ line %d\n", __FILE__, lineNum);
```

```
  strcpy(errorCode, "unknown error");
  switch (ret) {
    case CL_INVALID_KERNEL:
      strcpy(errorCode, "CL_INVALID_KERNEL");
      break;
    case CL_INVALID_ARG_INDEX:
      strcpy(errorCode, "CL_INVALID_ARG_INDEX");
      break;
    case CL_INVALID_ARG_VALUE:
      strcpy(errorCode, "CL_INVALID_ARG_VALUE");
      break;
    case CL_INVALID_MEM_OBJECT:
      strcpy(errorCode, "CL_INVALID_MEM_OBJECT");
      break;
    case CL_INVALID_SAMPLER:
      strcpy(errorCode, "CL_INVALID_SAMPLER");
      break;
    case CL_INVALID_ARG_SIZE:
      strcpy(errorCode, "CL_INVALID_ARG_SIZE");
      break;
  }

  fprintf(stderr, "clSetKernelArg %s\n", errorCode);
  exit(1);
  return 0;
}



/** Compiles and runs. Call cl_host_startup, cl_create_buffer and cl_buffer_write before this

int numTransforms: number of transforms to be run, also the number of kernels called
int loop_base: TODO either do this in fortran or in this function (set to 0 for now)

**/
int cl_host_run(int numTransforms, int loop_base){


  int i;

  /** Create Kernel Program from the source */
  program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
                                                    (const size_t *)&source_size, &ret);
    if (!program)
    {
          fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -4);
        printf("clCreateProgramWithSource failed to create compute program!\n");
        exit(1);
    }


  /** Build Kernel Program */
  ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      // Shows the log
      char* build_log;
      size_t log_size;
      // First call to know the proper size
      clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 0, NULL, &log_size);
      build_log = new char[log_size+1];
      // Second call to get the log
      clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, log_size, build_log, NULL);
      build_log[log_size] = '\0';

      fp = fopen ("opencl_build_errors_log.txt", "wt");
      fprintf(fp, ">>:%s\n", build_log);
//      printf(">>:%s\n", build_log);
      fclose(fp);

      fprintf(stderr, "Error: Failed to build program executable!\n");
      fprintf(stderr, "See opencl_build_errors_log.txt for build log\n");

      exit(1);

  }

  fprintf(stderr, "Kernel compiled\n");

  /** Create OpenCL Kernel */
  kernel = clCreateKernel(program, "fasten_main", &ret);
  if (ret != CL_SUCCESS)
  {
      fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
      fprintf(stderr, "clCreateKernel failed\n");
      exit(1);
  }

  //scalars
  ret = clSetKernelArg(kernel, 0, sizeof(cl_int), (void *)&natlig);
  if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
  ret = clSetKernelArg(kernel, 1, sizeof(cl_int), (void *)&natpro);
  if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
  ret = clSetKernelArg(kernel, 2, sizeof(cl_int), (void *)&ntransforms);
  if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
  ret = clSetKernelArg(kernel, 3, sizeof(cl_float), (void *)&cutdis);
```

```
if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);

//others
ret = clSetKernelArg(kernel, 4, sizeof(cl_mem), (void *)&memobj_prot_mol);
if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
ret = clSetKernelArg(kernel, 5, sizeof(cl_mem), (void *)&memobj_lig_mol);
if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
ret = clSetKernelArg(kernel, 6, sizeof(cl_mem), (void *)&memobj_trans);
if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);
ret = clSetKernelArg(kernel, 7, sizeof(cl_mem), (void *)&memobj_etotals);
if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);

size_t global_item_size = 1; //number of work items (globally)
size_t local_item_size = 1; //number of work groups

fprintf(stderr, "queueing kernel %d times\n", numTransforms);

int loopIncrement = 0;
int transformsRemaining = numTransforms;

for (int lbase=0;lbase<numTransforms;lbase += loopIncrement){

  transformsRemaining = numTransforms-lbase;

  /** If there are sufficient transformsRemaining, global_item_size and loopIncrement should be 40x cl_mwg_size  **/
  if(transformsRemaining > (cl_mwg_size*40) ){
    local_item_size = cl_mwg_size;
    global_item_size = cl_mwg_size*40;
    loopIncrement = cl_mwg_size*40;
  }
  else{ /** transformsRemaining is less than cl_mwg_size*40  **/
    if(transformsRemaining > (cl_mwg_size*5)){ /** transformsRemaining is greater than cl_mwg_size*5  **/
        local_item_size = cl_mwg_size;
        global_item_size = cl_mwg_size*5;
        loopIncrement = cl_mwg_size*5;
    }
    else{ /** transformsRemaining is less than cl_mwg_size*10  **/
        if(transformsRemaining > (cl_mwg_size)){ /** transformsRemaining is less than cl_mwg_size  **/
                  local_item_size = cl_mwg_size;
                  global_item_size = cl_mwg_size;
                  loopIncrement = cl_mwg_size;
        }
        else{ /** transformsRemaining is less than cl_mwg_size  **/
                  local_item_size = 1;
                  global_item_size = transformsRemaining;
                  loopIncrement = transformsRemaining;
        }
    }
  }


  fprintf(stderr, "%d/%d\n", lbase, numTransforms);


  ret = clSetKernelArg(kernel, 8, sizeof(cl_int), (void *)&lbase);
  if (ret != CL_SUCCESS) setKernelArgError(ret, __LINE__ -1);


  /** Execute OpenCL kernel as data parallel */
  ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                                    &global_item_size, &local_item_size, 0, NULL, &ev);
  if (ret != CL_SUCCESS)
  {
        fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -4);
        strcpy(errorCode, "unknown error");
        switch (ret) {
          case CL_INVALID_PROGRAM_EXECUTABLE:
            strcpy(errorCode, "CL_INVALID_PROGRAM_EXECUTABLE");
            break;
          case CL_INVALID_COMMAND_QUEUE:
            strcpy(errorCode, "CL_INVALID_COMMAND_QUEUE");
            break;
          case CL_INVALID_KERNEL:
            strcpy(errorCode, "CL_INVALID_KERNEL");
            break;
          case CL_INVALID_CONTEXT:
            strcpy(errorCode, "CL_INVALID_CONTEXT");
            break;
          case CL_INVALID_KERNEL_ARGS:
            strcpy(errorCode, "CL_INVALID_KERNEL_ARGS");
            break;
          case CL_INVALID_WORK_DIMENSION:
            strcpy(errorCode, "CL_INVALID_WORK_DIMENSION");
            break;
          case CL_INVALID_WORK_GROUP_SIZE:
            strcpy(errorCode, "CL_INVALID_WORK_GROUP_SIZE");
            break;
          case CL_INVALID_WORK_ITEM_SIZE:
            strcpy(errorCode, "CL_INVALID_WORK_ITEM_SIZE");
            break;
          case CL_INVALID_GLOBAL_OFFSET:
            strcpy(errorCode, "CL_INVALID_GLOBAL_OFFSET");
            break;
          case CL_OUT_OF_RESOURCES:
            strcpy(errorCode, "CL_OUT_OF_RESOURCES");
            break;
          case CL_MEM_OBJECT_ALLOCATION_FAILURE:
            strcpy(errorCode, "CL_MEM_OBJECT_ALLOCATION_FAILURE");
            break;
```

```
            case CL_INVALID_EVENT_WAIT_LIST:
              strcpy(errorCode, "CL_INVALID_EVENT_WAIT_LIST");
              break;
            case CL_OUT_OF_HOST_MEMORY:
              strcpy(errorCode, "CL_OUT_OF_HOST_MEMORY");
              break;
          }
          fprintf(stderr, "clEnqueueNDRangeKernel: %s\n", errorCode);

          exit(1);

    }

    /** Wait for the kernel to finish */
    clWaitForEvents(1, &ev);

    ret = clFlush(command_queue);
    if (ret != CL_SUCCESS)
    {
          fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
          fprintf(stderr, "clFlush failed\n");
          exit(1);
    }

  }


  return 0;
}

/** Waits for kernel to finish executing then fetches result.

int argID: position of memory object in memobj array
int numBytes: size of memory object
void *ptr: where result will be written too

**/
int cl_fetch_result(int argID, int numBytes, void *ptr){

  /** Wait for the kernel to finish */
  clWaitForEvents(1, &ev);

  fprintf(stderr, "clWaitForEvents complete\n");

  /** Transfer result to host */
  ret = clEnqueueReadBuffer(command_queue, memobj_etotals, CL_TRUE, 0, numBytes, ptr, 0, NULL, NULL);
  if (ret != CL_SUCCESS)
  {
    fprintf(stderr, "Error in %s @ line %d\n", __FILE__, __LINE__ -3);
    strcpy(errorCode, "unknown error");
    switch (ret) {
      case CL_INVALID_COMMAND_QUEUE:
          strcpy(errorCode, "CL_INVALID_COMMAND_QUEUE");
          break;
      case CL_INVALID_CONTEXT:
          strcpy(errorCode, "CL_INVALID_CONTEXT");
          break;
      case CL_INVALID_MEM_OBJECT:
          strcpy(errorCode, "CL_INVALID_MEM_OBJECT");
          break;
      case CL_INVALID_VALUE:
          strcpy(errorCode, "CL_INVALID_VALUE");
          break;
      case CL_INVALID_EVENT_WAIT_LIST:
          strcpy(errorCode, "CL_INVALID_EVENT_WAIT_LIST");
          break;
      case CL_MEM_OBJECT_ALLOCATION_FAILURE:
          strcpy(errorCode, "CL_MEM_OBJECT_ALLOCATION_FAILURE");
          break;
      case CL_OUT_OF_HOST_MEMORY:
          strcpy(errorCode, "CL_OUT_OF_HOST_MEMORY");
          break;
    }
    fprintf(stderr, "clEnqueueReadBuffer: %s\n", errorCode);

    exit(1);
  }

  return 0;
}


int cl_host_finalize(){

  int i;

  /** Finalization */
  ret = clFlush(command_queue);
  ret = clFinish(command_queue);
  ret = clReleaseKernel(kernel);
  ret = clReleaseProgram(program);
  ret = clReleaseMemObject(memobj_prot_mol);
  ret = clReleaseMemObject(memobj_lig_mol);
  ret = clReleaseMemObject(memobj_trans);
  ret = clReleaseMemObject(memobj_etotals);
  ret = clReleaseEvent(ev);
  ret = clReleaseCommandQueue(command_queue);
  ret = clReleaseContext(context);
```

```
  free(source_str);

  return 0;
}
```

bude_cl_calgen_rot.f

```
      subroutine calgen(info,kdeb,indat,iodat,ncnf,ig,kroute,xatlig,
     1    natlig,xatpro,natpro,transf,trsdec,ntrnsf,rmsbuf,enbuff,
     2    tilttr,rolltr,pantr,xtratr,ytratr,ztratr,rtcntx,rtcnty,rtcntz,
     3    hbtypp,rad_p,hphb_p,hard_p,elsc_p,nndstp,npdstp,atom_p,
     4    hbtypl,rad_l,hphb_l,hard_l,elsc_l,nndstl,npdstl,atom_l,
     5    srtval,fmix,krmsty,xattrg,cutdis,iuplig)
c
c-----Calculates the property values of a complete generation.
c     Transformation descriptors are held in core memory, and written to outfil.
c
      implicit NONE
c
      include "sizes.inc"
      include "constants.inc"
c
      integer info,kdeb,indat,iodat,ncnf,icnf,kroute,ig,natlig,natpro,

     1 trsdec(MXGSIZ,MXTRSF),ntrnsf,tilttr,rolltr,pantr,xtratr,ytratr,
     2 ztratr,krmsty,j,iuplig

      real xatlig(3,MXATM),xatpro(3,MXATM),transf(MXSTAT,MXTRSF),
     1  xlgbll(3,MXATM),rtcntx,rtcnty,rtcntz,etot,enbuff(MXGSIZ),
     2  rmstot,rmsbuf(MXGSIZ),srtval(MXGSIZ),fmix,xattrg(3,MXATM),
     3  cutdis

c-----The following vars hold forcefield properites for each protein/ligand ball
      real rad_p(MXATM),hard_p(MXATM),nndstp(MXATM),npdstp(MXATM),
     1  elsc_p(MXATM),hphb_p(MXATM),
     2  rad_l(MXATM),hard_l(MXATM),nndstl(MXATM),npdstl(MXATM),
     3  elsc_l(MXATM),hphb_l(MXATM)
      character*1 hbtypp(MXATM),hbtypl(MXATM)
      character*3 atom_p(MXATM),atom_l(MXATM)

      integer numkernels, loopbase

      real times(2,32)
      common /tim/ times
      double precision  secs0(2),secs1(2)

c-----------------------------------------
c The additional arrays needed for coprocessing
      character*40 ligand_molecule(MXATM)
      character*40 protein_molecule(MXATM)
      real         transforms(12,MXGSIZ)
c      real         stats(10,-1:32)               ! performance timers
c      real         offload                       ! what % of work to coprocess
      !integer*4    sizeof_atom
c      integer*4    SEM_GO,SEM_DONE
      logical      firstpass
      integer      ipe,npes, ibase,iend,nrem,ntransforms, ncnf_offload
      integer      ifail, i, verbose
      character*40 molecule,  envvar*255
      common       /cs_molecule/ molecule
      double precision  secs2(2),secs3(2)
c      data         npes/2/,offload/100./
c      data         SEM_GO,SEM_DONE/40,41/
      data         firstpass/.true./
c-----------------------------------------


      write(info,*) 'ncnf= ',ncnf,' kroute=',kroute

      ntransforms = ncnf


          write(info,*) 'INIT COMPLETE '
!-- Build then send the Protein and Ligand molecules
!-- Turn the set of arrays that define the protein molecule into one object
       if (firstpass) then
         firstpass=.false.
         do i=1,natpro
           call packat (xatpro(1,i), rad_p(i), hphb_p(i), hard_p(i),
     +        nndstp(i), npdstp(i), elsc_p(i), hbtypp(i), atom_p(i) )
          protein_molecule(i)= molecule
         enddo

!-- likewise for the ligand
         do i=1,natlig
           call packat (xatlig(1,i), rad_l(i), hphb_l(i), hard_l(i),
     +        nndstl(i), npdstl(i), elsc_l(i), hbtypl(i), atom_l(i) )
          ligand_molecule(i)= molecule
         enddo

            call cl_new_buf_f(3, 4)
            call cl_buffer_write_f(3, 4, cutdis)

            call cl_new_buf_i(1, 4)
            call cl_buffer_write_i(1, 4, natpro)
```

```
            call cl_new_buf(4, 4*40*natpro)
            call cl_buffer_write(4, 4*40*natpro, protein_molecule)

            call cl_new_buf_i(0, 4)
            call cl_buffer_write_i(0, 4, natlig)

            call cl_new_buf(5, 4*40*natlig)
            call cl_buffer_write(5, 40*natlig, ligand_molecule)

c           read buffer,
            call cl_new_buf_f(7, ncnf*4)

c           ntransforms:
            call cl_new_buf_i(2, 4)

c           transforms:
            call cl_new_buf_f(6, 4*12*ntransforms)

c      endif
       endif  ! if first pass AND coprocessing


c
c-----Update the ligand coodinates for the next conformer
c
      if (iuplig .eq. 1 .and. .not. firstpass) then
           do i=1,natlig
              call packat (xatlig(1,i), rad_l(i), hphb_l(i), hard_l(i),
     +         nndstl(i), npdstl(i), elsc_l(i), hbtypl(i), atom_l(i) )
              ligand_molecule(i)= molecule
            enddo


              call cl_buffer_write_i(0, 4, natlig)
              call cl_buffer_write(5, 40*natlig, ligand_molecule)

         iuplig = 0
      endif
c
c-------Calculate protery values, as specified in ctrl file
      if (kroute .eq. 1 .or. kroute .eq. 3) then

        if (ntransforms.gt.0) then


            call secnd(secs0)
c                   WAS:
c            call cnftrf(info,kdeb,transf,trsdec,tilttr,rolltr,pantr,
c     +         xtratr,ytratr,ztratr,rtcntx,rtcnty,rtcntz,
c     +         ibase, ntransforms,     transforms)
c                   NOW: ibase = 0, ntransforms = ncnf
            call cnftrf(info,kdeb,transf,trsdec,tilttr,rolltr,pantr,
     +         xtratr,ytratr,ztratr,rtcntx,rtcnty,rtcntz,
     +         1, ntransforms,     transforms)

            call secnd(secs1)

            times(1,1) = times(1,1) + real(secs1(1)) - real(secs0(1))
            times(2,1) = times(2,1) + real(secs1(2)) - real(secs0(2))

c           ntransforms:
              call cl_buffer_write_i(2, 4, ntransforms)

c           transforms:
              call cl_buffer_write_f_p(6, 4*12*ntransforms, transforms)

              call cl_run(ntransforms, 0)

          endif

c-------------------------------------------------------------------------
c wait for the results to come back from the coprocessors, then process
c-------------------------------------------------------------------------


c         fill the read buffer with random values
            do i=1, ncnf
              enbuff(i) = -1.010101
            enddo

            call cl_read_buf(7, ncnf*4, enbuff)


          do i=1, ncnf
            srtval(i) = enbuff(i)   !- take a copy for later ranking
          enddo


      endif    !- if kroute =1 or 3



c-------------------------------------------------------------------------
c  Handle routes 2 and 3 here - no coprocessing as it is quick
      if (kroute .eq. 2 .or. kroute .eq. 3) then
        do icnf= ibase, ibase+ntransforms-1
c-------Generate new ligand co-ordinates for this transformation descriptor.
```

```
            call secnd(secs0)
            call cnfgeo(info,kdeb,xatlig,natlig,transf,trsdec,icnf,ntrnsf,
     1        tilttr,rolltr,pantr,xtratr,ytratr,ztratr,xlgbll,rtcntx,
     2        rtcnty, rtcntz)
            call secnd(secs1)
            times(1,1) = times(1,1) + real(secs1(1)) - real(secs0(1))
            times(2,1) = times(2,1) + real(secs1(2)) - real(secs0(2))
c
c---------Calculate RMS
            call secnd(secs0)
            call dorms(xattrg,xlgbll,natlig,rmstot,krmsty)
            call secnd(secs1)
c---------Store the RMS value in an array.
            rmsbuf(icnf) = rmstot
c---------If this is RMS only calc, then store RMS for sorting.
            if (kroute .eq. 2) then
              srtval(icnf) = rmsbuf(icnf)
c---------Else, if its a mixed RMS/energy calc, then store a mixed value to sort.
            else
              srtval(icnf) = fmix*srtval(icnf) + rmsbuf(icnf)
            endif
c
          enddo
        endif

c
        return
        end


        subroutine packat (xat,rad,hphb,hard,nndst,npdst,elsc,hbtyp,atom)
c
c       This gathers together the values of a molecule atom's into a single
c       40 byte piece - equivalent to a F90 derived type.
c
        implicit NONE
        real xat(3),rad, hphb, hard, nndst, npdst, elsc
        real xat1(3),rad1, hphb1, hard1, nndst1, npdst1, elsc1
        character hbtyp*1, atom*3
        character hbtyp1*1, atom1*3
        common     /cs_molecule/xat1,rad1, hphb1, hard1, nndst1, npdst1,
       +                        elsc1, hbtyp1, atom1
        xat1(1)= xat(1)
        xat1(2)= xat(2)
        xat1(3)= xat(3)
        rad1=rad
        hphb1=hphb
        hard1=hard
        nndst1=nndst
        npdst1=npdst
        elsc1=elsc
        hbtyp1=hbtyp
        atom1=atom
        return
        end


        subroutine cnftrf(info,kdeb,transf,trsdec,tilttr,rolltr,pantr,
       1 xtratr,ytratr,ztratr,rtcntx,rtcnty,rtcntz,
       +      ibase, ntransforms,   transforms)
c
c This calculates the transformation matrices for ntrans transformation
c descriptors (TDs). The result is returned in trfmat(12,MXMTAP)
c
        implicit NONE
        include "sizes.inc"
        include "constants.inc"
        integer info,kdeb,nat,trsdec(MXGSIZ,MXTRSF),icnf,itrans,
       1  ntrnsf,tilttr,rolltr,pantr,xtratr,ytratr,ztratr,i,ntrans,
       2  ipf,ipl,ibase, ntransforms

        real transf(MXSTAT,MXTRSF)                ! database of trial values
        real rtcntx,rtcnty,rtcntz                 ! xyz shift
        real xa,ya,za,xt,yt,zt,cx,sx,cy,sy,cz,sz  ! temporaries
        real transforms(12,MXGSIZ)                ! output of 3x4 matrices

        !print*,' 6 trs are', tilttr,rolltr,pantr,xtratr,ytratr,ztratr
        !print*,' 3 shifts are ', rtcntx,rtcnty,rtcntz
c-----Fill the buffer trfmat with ntrans transformation matrices
        do i=1,ntransforms
           icnf = ibase + i - 1
c--------get the rotation and translation values for this TD
           xa = transf (trsdec(icnf,tilttr),tilttr)
           ya = transf (trsdec(icnf,rolltr),rolltr)
           za = transf (trsdec(icnf,pantr),pantr)
           xt = transf (trsdec(icnf,xtratr),xtratr)
           yt = transf (trsdec(icnf,ytratr),ytratr)
           zt = transf (trsdec(icnf,ztratr),ztratr)
c--------add in the translation components to move the ligand back from
c          the origin where it was moved to do the rotations...
           xt = xt + rtcntx
           yt = yt + rtcnty
           zt = zt + rtcntz
c--------do the sines and cosines (should really do this to the transf
c          contents during setup - ie generate the cos & sines only once)
           cx = cos(xa)
           sx = sin(xa)
           cy = cos(ya)
           sy = sin(ya)
```

```
            cz = cos(za)
            sz = sin(za)
c--------Now do the maths to multiply the 3 rotation matrices for all three
c           rotations and store the result and the translations into trfmat
            transforms( 1,i) = cy*cz
            transforms( 2,i) = sx*sy*cz - cx*sz
            transforms( 3,i) = cx*sy*cz + sx*sz
            transforms( 4,i) = xt
            transforms( 5,i) = cy*sz
            transforms( 6,i) = sx*sy*sz + cx*cz
            transforms( 7,i) = cx*sy*sz - sx*cz
            transforms( 8,i) = yt
            transforms( 9,i) = -sy
            transforms(10,i) = sx*cy
            transforms(11,i) = cx*cy
            transforms(12,i) = zt
         enddo
         return
         end
c
c
```