Map-reduce as a Programming Model for Custom Computing Machines

Jackson H.C. Yeung[1], C.C. Tsang[1], K.H. Tsoi[1], Bill S.H. Kwan[1],
Chris C.C. Cheung[2], Anthony P.C. Chan[2] and Philip H.W. Leong[1]

[1]Dept. of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin NT, Hong Kong
and
[2]Cluster Technology Limited
Hong Kong Science and Technology Park, NT, Hong Kong

{hcyeung,cctsang,khtsoi,bkwan,phwl}@cse.cuhk.edu.hk, {cheungcc,pcchan}@clustertech.com

Abstract

The map-reduce model requires users to express their problem in terms of a map function that processes single records in a stream, and a reduce function that merges all mapped outputs to produce a final result. By exposing structural similarity in this way, a number of key issues associated with the design of custom computing machines including parallelisation; design complexity; software-hardware partitioning; hardware-dependency, portability and scalability can be easily addressed.

We present an implementation of a map-reduce library supporting parallel field programmable gate arrays (FPGAs) and graphics processing units (GPUs). Parallelisation due to pipelining, multiple datapaths and concurrent execution of FPGA/GPU hardware is automatically achieved. Users first specify the map and reduce steps for the problem in ANSI C and no knowledge of the underlying hardware or parallelisation is needed. The source code is then manually translated into a pipelined datapath which, along with the map-reduce library, is compiled into appropriate binary configurations for the processing units. We describe our experience in developing a number of benchmark problems in signal processing, Monte Carlo simulation and scientific computing as well as report on the performance of FPGA, GPU and hetereogeneous systems.

1 Introduction

Reconfigurable computing has been successfully applied to a diverse range of applications including sorting and searching, signal processing, cryptography, scientific computing and logic emulation. High performance is obtained by achieving high degrees of parallelism, and problem-specific customisation of the hardware can be carried out to a degree not possible in other technologies.

Extracting parallelism is a key issue in the design of such machines. In the ideal case, designs would be described in a sequential, algorithmic fashion and parallelism would be automatically extracted by a compiler. Despite decades of research in this area for both parallel computers and field programmable custom computing machines (FCCMs), satisfactory tools still do not exist. Common practice is to identify the hardware/software interface and the hardware parallelism manually, and then design separate datapath and control circuits to implement them using reconfigurable fabric. This ad-hoc approach is used often to the detriment of design time, efficiency, portability and reuse.

In this work, a parallelisation methodology based on the map-reduce higher order functions common in functional languages is presented which supports both field programmable gate array (FPGA) and graphics processing unit (GPU) based processing units. Map-reduce has the following advantages:

- When a problem is expressed in a map-reduce form, it is easy to parallelise the computation, distribute data to the processors and to load balance between them. The details concerning all these issues can be hidden from the user and opportunities for task-level and instruction-level parallelisation are easily identified.
- Designs are easily partitioned between hardware and software.
- Map-reduce provides an interface that is independent of the back-end technology. This provides a conve-

nient means for employing multiple, heterogeneous accelerators; separate machine dependent and machine independent implementation issues and improve portability.

- The complexity of a map-reduce library is very low and can be understood, modified and extended by most developers. This is in contrast to a parallelising compiler which requires far more expertise.

Although functional programming languages are well known to be good for expressing parallelism, we are not aware of any other design methodology for reconfigurable computing that is able to combine the benefits of the proposed approach. Map-reduce can be used in conjunction with other techniques to further improve performance.

The rest of the paper is organised as follows. In Section 2, a review of previous map-reduce software implementations is given. In Section 3, our map-reduce methodology is detailed. Our benchmark set is introduced in Section 4 and results are given in Section 5. Finally, conclusions are drawn in Section 6.

2 Background

The origins of the map function in programming can be traced back to the LISP programming language [1] and reduce to APL [2], the precise specification being dependent on the implementation. A detailed study of several different implementations is given in reference [3]. Map-reduce operations are often used in standard imperative languages. Waters studied programs in the IBM Scientific Subroutine Package and found that that 90% of the code could be expressed as maps, filters and accumulations [4].

Map-reduce typically takes as inputs: a list of input records *l*, a map function and a reduce function. The map function is applied to each element of the list to form a new list to which the binary reduce operation is then applied. We assume the map and reduce functions do not have side effects, and reduce is a binary operator which is both associative and commutative. This means that the map and reduce operations can be executed in any order and in parallel. In addition, we assume that the list can be infinite in length, and is hence a stream.

We will use a simple Monte Carlo simulation (MCS) to compute an approximation to π as an example of applying map-reduce. Imagine a circle of radius r circumscribed by a square with sides of length $2r$. If a large number of darts are thrown uniformly at the square, the proportion of darts which hit inside the circle is given by:

$$\frac{\text{area of circle}}{\text{area of square}} = \frac{\pi r^2}{(2r)^2} = \pi/4. \tag{1}$$

The above proportion is the same if only the top, right hand quarter of a square centered at the origin is considered. Thus, if $r = 1$, π can be approximated by randomly generating two random numbers, $x, y \in [0, 1)$, calculating whether the coordinate (x, y) is within the top quarter of a circle ($x^2 + y^2 < 1$), counting the proportion of trials inside and outside the circle, and multiplying this result by 4.

Using map-reduce, the map function, pi_map, is

$$pi\_map(x, y) = \begin{cases} 1 & ((x^2 + y^2) < 1) \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

and the reduce function

$$pi\_reduce(a, b) = a + b. \tag{3}$$

A stream of N pairs of uniform random numbers $\in [0, 1)$ are applied to the map function and the outputs reduced to produce the value of h in the above pseudocode. The approximation is then computed as $\frac{4h}{N}$.

Map-reduce is used in Google's "MapReduce" library to utilise large-scale clusters for parallelised data processing applications [5]. Programmers simply describe the associated map-reduce computation and a map-reduce library deals with the issues of configuration, initialisation, networking, load balancing and fault tolerance. This serves to provide programmers with a simple means to develop applications for a massively parallel machine without the usual associated complexity.

Sawzall is an interpreted language used at Google which makes writing MapReduce programs easier and 10-20 times shorter. For example in March 2005, on a 1500 machine cluster at Google, 32,580 Sawzall jobs were launched, each using an average of 220 machines. In total, 2.8 pentabytes of data were read, 9.3 terabytes written and one machine-century of central processing unit time were consumed [6].

Chu et. al., used map-reduce as a framework for implementing parallelised machine learning algorithms on multicore machines [7]. They showed that a variety of algorithms including locally weighted linear regression, K-means, logistic regression, naive Bayes, support vector machine, independent component analysis, principal component analysis, Gaussian discriminative analysis, expectation maximisation and backpropagation can be described and efficiently implemented on multicore and multiprocessor machines. The Brook [8] language for stream computing on GPUs directly supports the efficient compilation of map and reduce operations.

3 Map-reduce Methodology

3.1 API

The application programming interface (API) of the map-reduce implementation has a single entry point,

mapreduce() and the mapreduce_t structure provides all of the required data.

```
typedef struct {
  void *parameter; /* passed to fns */
  int  i_size; /* in rec size (bytes) */
  int  o_size; /* out rec size */
  int  (*infn)(void *i_buf, void *param);
  void (*mapfn)(void *o_buf,
          void *ibuf, void *param);
  void (*redfn)(void *result, void *o_buf,
          void *param);
  void *result; /* result written here */
} mapreduce_t;

extern void mapreduce(mapreduce_t *);
```

All pointers to input and output records require preallocated memory for a single record entry. The parameter argument is used to pass information to the three function pointers infn, mapfn and redfn. This is normally a fixed block of memory which is directly copied from the host machine to the hardware accelerator card for access during the computation.

Since we assume map-reduce functions do not have side-effects, multiple instances can be executed in parallel without conflict. Unfortunately, strictly conforming is restrictive and in fact, the API already has side-effects as the map function must put its result in the o_buf buffer and the reduce function manipulates the result pointer *result. As an example, the π calculation described in Section 2 makes a function call to a random number generator (RNG) which requires state information. If parallel instances are invoked with the same initial state, identical sequences would result. Our solution is to enforce the rule that the map and reduce functions have no side effects with the exception that: a library of safe functions for random number generation can be called and writing to o_buf and *result are allowed.

3.2 Software Implementation

An example of the usage of this library for the π example is given in this section. The user must first manually partition the algorithm into input, map and reduce functions. The code for these functions are described below.

First the input and output types of the map function are defined.

```
typedef struct {
  float a; float b;
} map_in_t;

typedef int map_out_t;

typedef struct {
  unsigned iterations;
} map_param_t;
```

Then the three required functions are described as below.

```
int
pi_input(map_in_t *e, map_param_t *p) {
  e->a = Tausworthe_RAND();
  e->b = Tausworthe_RAND();
  /* return 0 when finished */
  return p->iterations--;
}

void
pi_map(map_out_t *out,
  map_in_t *in, map_param_t *p) {
  *out = (in->a*in->a + in->b*in->b) > 1.0
         ? 0 : 1;
}

void
pi_reduce(map_out_t *r,
  map_out_t *a, map_param_t *p ) {
  *r = *r + *a;
}
```

The pi_input() function generates the two random numbers for the map function to consume. It uses the library's builtin uniform random number generator which implements the Tausworthe algorithm [9]. This RNG initialises state information uniquely for different map executions and hence can safely be invoked in parallel. Other library functions may also be used to access special features of the hardware. These functions are replaced by the appropriate implementation on the target platform. The pi_map() and pi_reduce() functions implement Equation 2 and Equation 3 respectively in a straightforward manner. This implementation of pi_reduce only takes one input and hence does not allow an efficient binary reduction tree implementation. This can be improved in future versions.

To supply the required parameters to the mapreduce() function, the user creates a mapreduce_t structure and fills in the relevant fields.

```
mapreduce_t m;
map_out_t result;

map_param_t p;
m.parameter = &p;
m.i_size = sizeof( map_in_t );
m.o_size = sizeof( map_out_t );
m.infn = &pi_input;
m.mapfn = &pi_map;
m.redfn = &pi_reduce;
m.result = &result
```

Finally a call to mapreduce (mapreduce(&m)) is made which performs the calculation. The result of the computation is stored in the location pointed to by result.

The above implementation can be translated to the target platform as described in the next section. A C version of the `mapreduce()` function is given below.

```
void mapreduce( mapreduce_t *mp ) {
  int   i;
  void *i_buf;
  void *o_buf;

  /* allocate the memory */
  i_buf = malloc( mp->i_size );
  o_buf = malloc( mp->o_size );

  while ((*mp->infn)(i_buf,
        mp->parameter, i, NULL, 0)) {
    (*mp->mapfn)
        (o_buf, i_buf, mp->parameter, i);
    (*mp->redfn)
        (mp->result, o_buf, mp->parameter);
  }
  free( i_buf ); free( o_buf );
}
```

3.3 Hardware Translation

The next phase is to translate the above code to the target hardware, this being simplified because the functions do not have side effects. Although translation is currently done manually, we are working on a simple source-to-source compiler that takes the ANSI C input, map and reduce functions, and creates a Celoxica Handel-C implementation. The HyperStreams library [10] is used to create pipelined implementations of the basic blocks and control flow is handled in Handel-C. These descriptions can then be directly compiled to FPGA bitstreams. If the functions have no dependencies, as is the case for the π example, the resulting FPGA implementation is fully pipelined, an output being produced every cycle. We note that the quality of the implementation generated by Handel-C can greatly affect the overall performance of the system.

```
macro proc pi_map( Output, X, Y, Domain ) {
  HS_SINGLE(One); HS_SINGLE(X2);
  HS_SINGLE(Y2); HS_SINGLE(R2);
  HS_BOOL(R2LtOne);

  par {
    HsSyncConstant(&One, 1.0);
    /* match latency of the 3 streams */
    HsSync(3, { X, Y, &One }, Domain);
    HsMul(X, X, &X2); HsMul(Y, Y, &Y2);
    HsAdd(&X2, &Y2, &R2);
    HsLt(&R2, &One, &R2LtOne);
    HsConvert( &R2LtOne, Output); }
}

macro proc
pi_reduce(Input, N, Output, Domain)
{ HsSumVar(Input, N, Sum, Domain); }
```

Multiple map-reduce pipelines are instantiated on the same FPGA in order to further increase parallelism. A single core is first created and its resource utilisation measured. The maximum number of cores that can fit on the FPGA is then estimated and a new design generated. In contrast to some other parallelisation schemes, this ensures that most of the resources on the FPGA are utilised.

3.4 Graphics Processing Unit

The NVIDIA Geforce 8 series GPUs contain multiple SIMD processors. Each "multiprocessor" has 8 SIMD processors, a small (16 kB) user programmable cache, an instruction unit, register file and local memory. Using NVIDIA's CUDA C-to-GPU compiler [11], GPU programs can be written in a similar way to standard multithreaded C programs.

Apart from minor differences in the SIMD model, host to hardware data transfer, memory model and language syntax, the CUDA code is analogous to a fixed hardware pipeline in HyperStreams, while multithreading is analogous to building multiple cores on an FPGA. As a result, the methodology for translating the input, map and reduce functions from C to HyperStreams or C to CUDA code is similar, and the actual body of the map-reduce C descriptions can be used directly in the GPU implementation.

3.5 Scheduling

Parallel map-reduce operations are performed on heterogeneous processing units (either FPGA or GPU) in our implementation. A POSIX pthreads-based multi-threaded scheduler is employed, allowing us to make use of multicore processor technology. The software-based scheduler is responsible for data buffering, task execution and load-balancing between computational threads. The computational threads supply data to hardware processing units which can be an arbitrary mix of FPGA and GPU boards.

In the current implementation, the input data are first divided by the scheduler into a number of temporary buffers. This is done by calling the input function the appropriate number of times to fill the buffers. This subtask data is then streamed to each parallel processing unit which performs the map-reduce operation and returns a reduced result. When a subtask is completed, the computational thread will reduce its output with previous outputs and assign a new subtask. This is repeated until the entire computation has completed. A diagram illustrating this process is given in Figure 1. Although not currently implemented, dynamic subtask sizing in which fast processing units get larger jobs may improve performance.

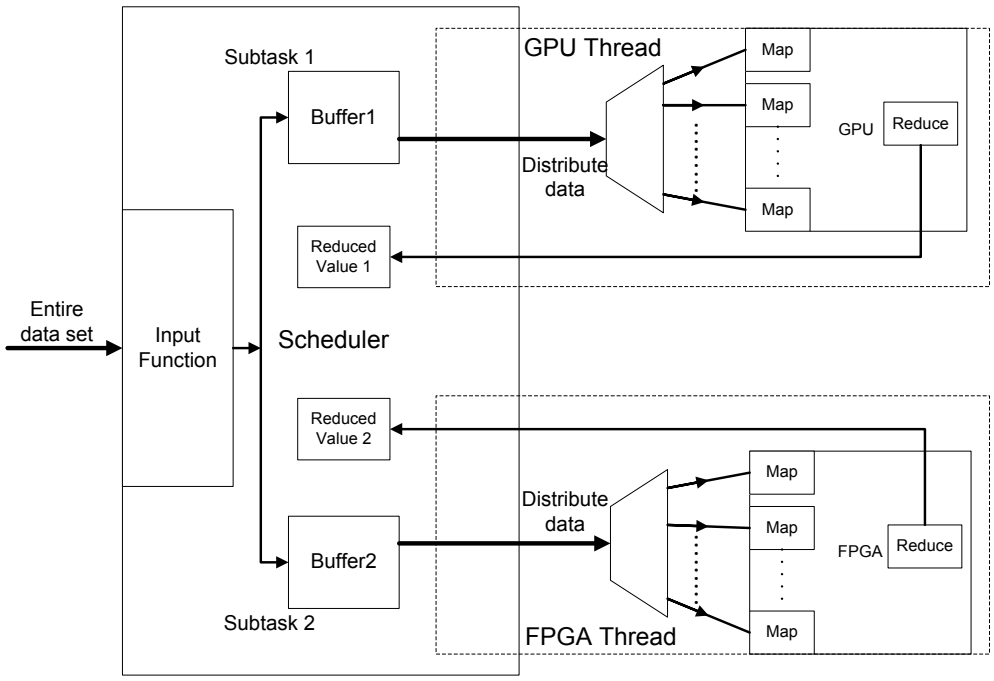

Figure 1. Illustration of the operation of the scheduler.

Although the above is a clean model for hardware-software partitioning, it may not result in the highest efficiency. As an example, the input function is best generated in hardware as its computation takes a significant fraction of total execution time, and sending a large amount of data from a software-based input function to a hardware-based map-reduce function is a bottleneck. A solution is an optimisation in which the input function is merged with the map function as shown below.

```
void
pi_fastmap(map_out_t *out, map_param_t *p) {
  float a = Tausworthe_Rand();
  float b = Tausworthe_Rand();
  *out = (a*a + b*b) > 1.0 ? 0 : 1;
}
```

In order to use this new function, it must be called an appropriate (possibly infinite as the input could be a stream of data) number of times. This is done with an internal iterator function supplied in the library called `range()`, a special type of input function. In hardware, `range()` is implemented using a counter that feeds directly to `pi_fastmap()`. The initial and final values of this counter are all that needs be transferred from the host to the processing unit. Furthermore, in contrast to arbitrary loops, the semantics of `range()` are easy to understand so the scheduler can divide ranges into subranges and execute them in parallel. This is not possible for general input functions which often have side effects as they may need to read files, analogue-to-digital converters etc.

4 Benchmarks

To demonstrate the applicability of our map-reduce framework, five examples have been constructed. Each example is presented in detail in this section. We found the implementations of the benchmarks using map-reduce had very similar architectures to standard parallel implementations. It should be noted that in our implementations of the π, European option and N-body benchmarks, for very large numbers of iterations, inaccurate results are returned. This is because relatively small numbers are added to very large accumulated sums resulting in numerical errors. In the future we will employ an algorithm such as the Kahan summation algorithm [12] to improve accuracy.

Dot Product. The dot product $c = \sum_i a_i b_i$ is a primitive in many signal processing and linear algebra applications including filters, transforms and regression. In our map-reduce implementation, we perform the multiplications in the map function and additions in the reduce function. We note that we can express matrix-vector and matrix-matrix products in terms of the dot product by simply changing the data types of the records and operators. For example, in a matrix-vector product ($\mathbf{m}v$), the records are the rows of $\mathbf{m}$ and each record is mapped by a dot product with v.

For a hardware-based implementation of dot product, the I/O overheads are expected to far exceed any computational

advantages of a parallel datapath. Hence we made another implementation which reduces the I/O by using random input data which is generated in hardware.

π Computation. Monte Carlo simulation is used to approximate the value of π as presented in section 2. The implementation of the map function in this example includes a call to the Tausworthe RNG algorithm and the `range()` input function is used. For N paths, the output of the map function is a stream of binary values $\in \{0, 1\}$. The reduce function is addition and π is computed by multiplying the reduced value by $4/N$ on the host computer.

European Option. The Monte Carlo simulation of a Black Scholes options pricing model for a European call option [13, 14] is computed via Monte Carlo simulation. This benchmark is very similar to the π example and is described by the following pseudocode:

```
europt() {
  for k=1 to N {
    PriceVary = exp (Mean + SD * GRNG());
    MyPrice = Price * PriceVary;
    Profit = MyPrice - Strike;
    if (Profit > 0)
      PayoffSum = PayoffSum + Profit;
  }
  return PayoffSum;
}
```

where *GRNG()* is a function call to a Gaussian random number generator (GRNG) implemented using the Box Muller method [15]. In this example, the map function includes the GRNG and the calculation of `Profit` and implements a single path in the simulation. As for the π computation, the `range()` input function is used. The output of the map function is the computed value of `Profit` in which negative values are replaced by zero. The reduce function sums the outputs of the map function. The pseudocode for the corresponding mapreduce implementation is given below.

```
euro_fastmap(out, param) {
  in = GRNG();
  PriceVary = exp(param.Mean +
    param.SD * in);
  out = param.Price * PriceVary -
    param.Strike;
}

euro_reduce(result, in, param) {
  if (in > 0)
      result = result + in;
}
```

RC4 Key Search. Our map-reduce framework is also used to parallelise a known plaintext attack of a 16-byte message using the RC4 cipher [16]. The possible key space of the 40-bit password is divided into blocks with a starting key. The map function input is an index indicating the position to start the search and implemented using `range()`. The output is 1 and the key if the search was successful, zero otherwise. Each map function performs a key scheduling process and generates a stream which is compared to the know sequence. The map function is shown below, all operations being byte operations:

```
for all keys in assigned search space {
  /* key initialisation */
  for i=0 to 255 state[i] = i;

  /* key scheduling */
  j = 0;
  for i=0 to 255 {
    j = (j + key[i] + state[i]);
    swap state[i] stage[j];
  }

  /* stream phase */
  i = 0; j = 0;
  for k=0 to TXT_LENGTH-1 {
    i = i + 1;
    j = j + state[i];
    swap state[i] state[j];
    t = state[i] + state[j];
    ctxt[k] = state[t];
  }

  /* compare to known sequence */
  if ctxt == KNOWN_SEQ
    return {1, key};
  else
    return {0, NULL};
}
```

This problem differs from the others in that it contains loops with dependencies. As a result, a lower degree of pipelining can be achieved. The reduce function, implemented on the host, checks the return value and outputs the correct key if found.

N-body Problem. In this example, our map-reduce framework was applied to the n-body simulation which traces the trajectory in time of n particles under gravitation force [17]. In our implementation, $n = 16384$ and initialisation of the particles was randomly generated using the Tausworthe RNG. Input to the map function are the current information for the n particles (passed in the `*param` pointer) and the particle index to be computed. The output is its acceleration. Hence for each particle, acceleration is computed as follows:

$$\mathbf{a_j} = \sum_{k \neq j}^{n} m_k \frac{(\mathbf{x_j} - \mathbf{x_k})}{|\mathbf{x_j} - \mathbf{x_k}|^3} \; j = 1, 2, \ldots, n \qquad (4)$$

and the new state for each particle is computed in the reduce function:

$$\begin{aligned} \mathbf{v_j} &= \mathbf{v_j} + \mathbf{a_j}\Delta t \\ \mathbf{x_j} &= \mathbf{x_j} + \mathbf{v_j}\Delta t \end{aligned}$$

where $\mathbf{a_j}$, $\mathbf{v_j}$ and $\mathbf{x_j}$ are the acceleration, velocity and position vectors for particle j respectively, m_j is its mass and Δt is a constant timestep.

5 Results

This section describes results obtained running the benchmarks described in the previous section. C source code is compiled with gcc 4.1.1 (-O3 optimisation) and used as a baseline for comparison. All FPGA implementations are compiled using Celoxica Handel-C DK5 and implemented using Xilinx ISE9.2i. GPU implementations are compiled using CUDA toolkit 1.1.

All tests are run on a personal computer (PC) equipped with a 2.4 GHz Intel Core 2 Duo E6600 central processing unit (CPU) and 2 GB of main memory running Linux. A Celoxica RC2000-Pro board (with a Xilinx Virtex II Pro XC2VP100-5 device) and an NVIDIA Geforce 8800GTX GPU are connected to the PC. The 8800GTX has 16 multiprocessors and has a total of 128 processing units. In all GPU implementations, 4096 threads are used, although results are not sensitive to this parameter.

The benchmark programs are implemented using the map-reduce methodology and library. This results in descriptions which are concise and a large amount of code reuse.

5.1 Multiple Pipelines

Figure 2 shows the speedup of the π benchmark compared with the CPU-based software implementation, for different numbers of simulation paths (iterations). The CPU implementation is not multithreaded so only a single core is used. For small numbers of paths, no speedup is achieved as the overhead of initialising and transferring data to the FPGA/GPU card does not justify the amount of computation to be performed. At 1 million paths, the FPGA speedup is approximately linear with the number of pipeline cores as expected. GPU initialisation has a higher overhead than the FPGA (of the order of hundreds of milliseconds) but for a large number of paths, its performance is greater. Even for a very large problem size, we were unable to saturate the GPU's performance for this simple computation. Of course, if problem size or complexity is further increased, we would expect saturation of the curve for this example as GPU resources are limited.

5.2 FPGA and GPU Comparison

FPGA and GPU implementations of the benchmarks were created. FPGA implementation details are given in Table 1 and the speedup in Figure 3, where speedup is the maximum speedup observed over a number of different problem sizes. In all cases, speedup increases with problem size as this increases the ratio of computation that can be performed in hardware compared with the initialisation and transfer overheads. In general, higher maximum speedups were observed for the GPU than the FPGA, especially for large problem sizes.

As expected, the dot product does not show any speedup over the CPU implementation as the overhead of transferring data to the FPGA or GPU card is very high. For the random dot product case, there is essentially no I/O and significant speedups are seen.

Due to the way iteration is handled in the translation process described in Section 3.3, the FPGA-based N-body implementation is not very efficient. The sum in Equation 4 is implemented as a loop and the rest as a parallel datapath and is not fully pipelined. This results in relatively poor performance compared with both the CPU and GPU. We expect that a better pipelining scheme for the FPGA would improve its performance by an order of magnitude.

For RC4, neither the GPU nor FPGA is able to achieve a significant speedup. This is partly because the RC4 algorithm runs extremely efficiently on the CPU, but also surprising as we reported on a manually-optimised FPGA implementation in 2002 which was able to achieve a $60\times$ improvement over a CPU [16]. Our manual implementation was very compact which allowed 96 cores to be implemented on a single FPGA. In contrast, our Handel-C implementation of the map function involved a direct translation from C, and only 5 cores could fit on a single FPGA; moreover, more clock cycles per key are also required and it runs at a lower clock frequency. We hence believe that the poor performance of the FPGA is due to Handel-C synthesising a less than optimal implementation of the algorithm and much better performance of a map-reduce implementation is achievable. Similarly, the GPU implementation could be better optimised for memory locality and thus performance.

The π and European option examples achieved significant speedups. Both involve single precision floating point computations and are fully pipelined, hence high degrees of parallelism achieved.

The speedup as a function of the problem size is shown for the European option and the N-body problem respectively in Figure 4 and Figure 5. In both cases, performance improves with problem size as initialisation and transfer overheads are amortised over a larger amount of computation. For the European option, the FPGA implementation is faster than the GPU for less than 40 million paths (we note

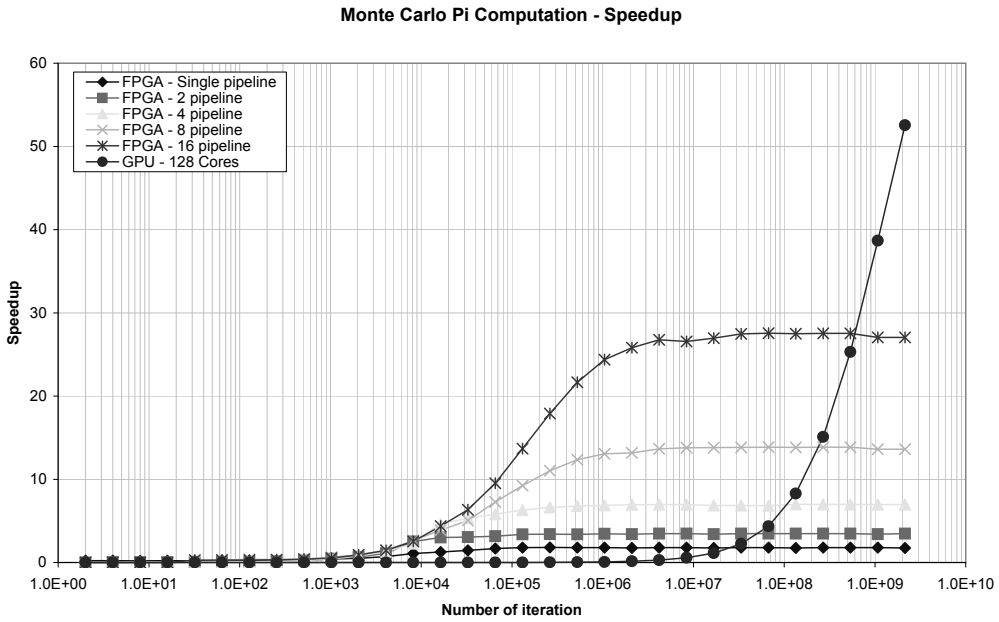

Figure 2. Monte Carlo π computation speedup.

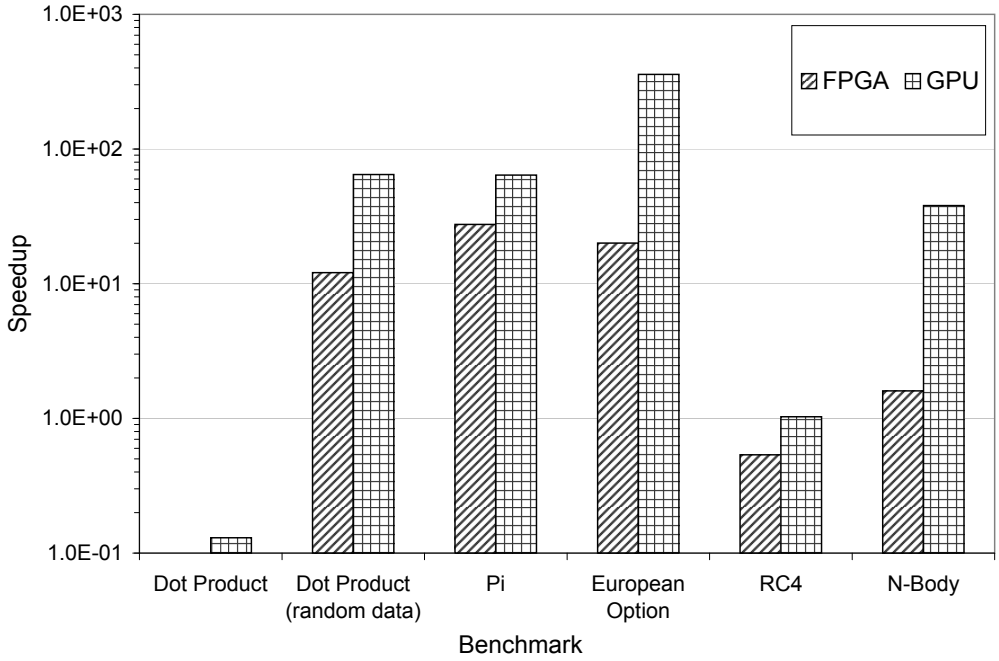

Figure 3. Speedup of FPGA and GPU compared with the CPU. Implementation details are given in Table 1, the largest problem size being chosen for each benchmark.

Bmark	Cores	Clock (MHz)	Area (Slices)	B-rams	Speed-up (×)
Dot	8	32	14542	32	7e-05
Dot (rand)	16	40	44094	16	12.1
π	1	63	3430	16	1.8
π	2	60	6053	16	3.5
π	4	61	11285	16	7.0
π	8	60	21590	16	13.9
π	16	60	42591	16	27.6
Euro	1	53	23106	78	20.0
RC4	1	40	10839	32	0.1
RC4	2	40	20823	32	0.2
RC4	5	40	44094	32	0.5
N-bdy	1	31	26098	435	1.6

Table 1. FPGA implementation and performance summary.

that in practical applications, one would not expect to require more than this number of paths) and hence the FPGA has higher speedup due to lower initialisation costs for small to medium sized problems. Remarkably, the speedup for the GPU only begins to saturate at 2^{31} paths.

5.3 Heterogeneous Execution

We also performed a test in which the benchmark examples are executed on both the FPGA and GPU simultaneously. The problem size for some of the benchmarks is reduced so that the FPGA and GPU speedup are similar in value and the results are shown in Figure 6. It can be seen that in cases where the FPGA and GPU performance is similar, heterogeneous execution results in an overall speedup over a single processing unit. For the benchmarks in the figure with much higher GPU speedup than FPGA speedup, the combined performance is very close to the GPU performance as the FPGA contributes little to the overall computing power. Best utilisation of a heterogeneous system would be for problems which can be divided into different parts where the GPU is efficient for one and the FPGA another.

6 Conclusion

A map-reduce library which encapsulates the code required for data transfer, interfacing and scheduling was demonstrated, along with a number of applications to signal processing, Monte Carlo simulation and scientific computing. It was further shown that, aided by the map-reduce methodology and library, sizeable systems could be developed with high productivity and using concise problem descriptions on heterogeneous FPGA/GPU-based custom computing machines.

Map-reduce restricts the programming model to some degree but offers the benefit that many difficult problems associated with the design and portability of systems can be greatly simplified. Excellent performance is achieved through parallelism due to pipelining of the operators, multiple cores and multiple GPU and FPGA processing units.

Overall, GPUs showed higher levels of speedup over FPGAs. This is partly due to the benchmark set consisting mostly of floating point and word-level operations for which the GPU is better optimised, but also because the FPGA platform used was much older than the GPU ($0.13\ \mu m$ vs $90\ nm$ technology). The programming effort required to produce the FPGA implementations was much higher than that of the GPU, mainly due to two factors: the very long compilation times (an hour for the FPGA compared with seconds for the GPU) and the fact that CUDA uses standard C with minor extensions whereas HyperStreams is significantly different from C requires manual matching of latencies in the datapath.

We believe that there is much scope for further research in this area. Issues we intend to study in the future include: power consumption; optimising compilers to efficiently translate map-reduce functions directly to CUDA and Handel-C; improved implementations of the map-reduce library to support more generalised scheduling and map-reduce operations; and other applications as case studies, particularly in the data-mining and scientific computing domains.

Acknowledgements

The authors gratefully acknowledge the support of the Hong Kong Innovation and Technology Grant ITS/027/07, "A Compiler for High Performance Computing on Array Technologies," and the Xilinx University Program.

References

[1] J. L. McCarthy, "Recursive functions of symbolic expressions and their computation by machine, part i," *Communications of the ACM*, vol. 3, no. 4, pp. 184–195, 1960.

[2] K. E. Iverson, *A programming language*. New York, NY, USA: John Wiley & Sons, Inc., 1962.

[3] R. Lämmel, "Google's MapReduce Programming Model – Revisited," 2007, to appear in Science of Computer Programming Journal; http://www.cs.vu.nl/ ralf/MapReduce/.

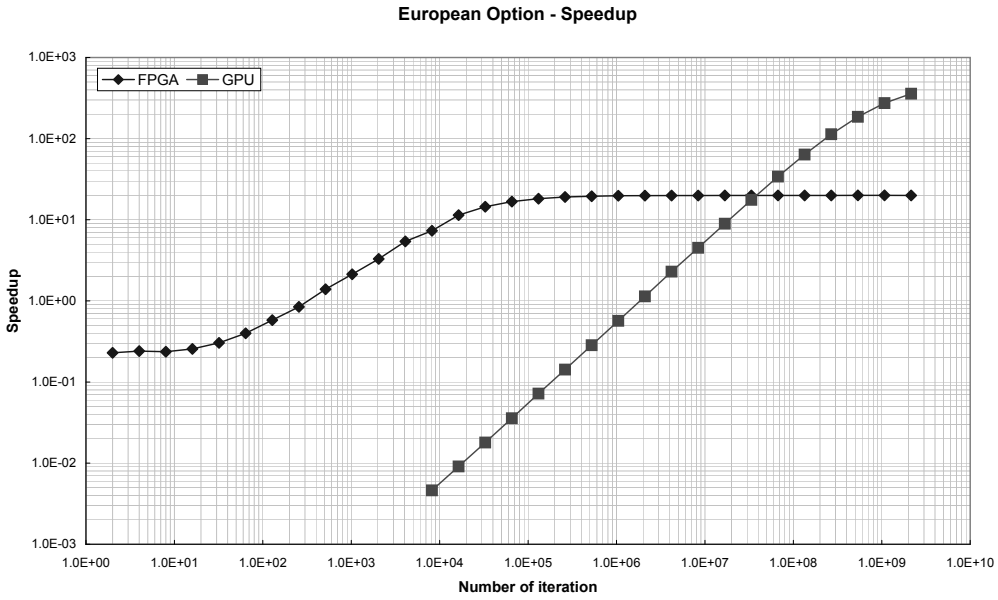

Figure 4. European option speedup.

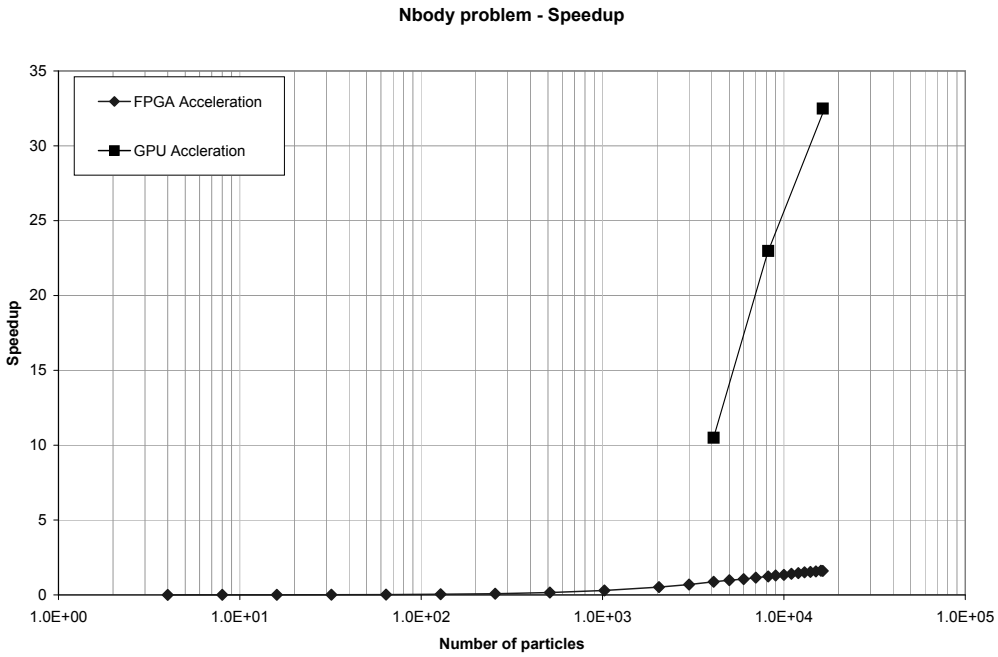

Figure 5. N-body problem speedup.

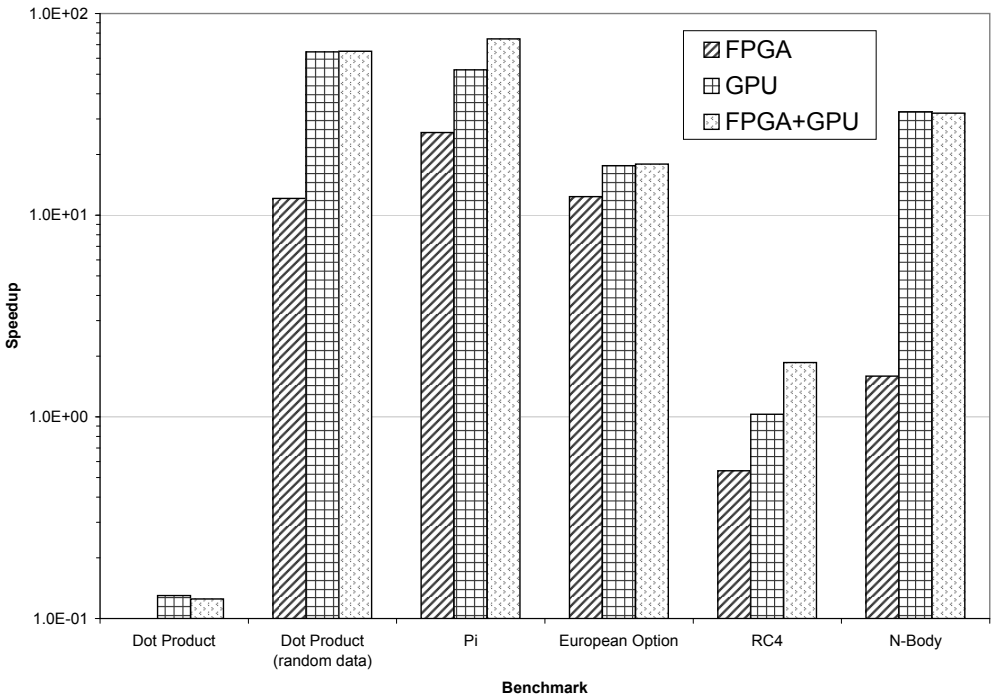

Figure 6. Speedup of heterogeneous system.

[4] R. C. Waters, "A method for analyzing loop programs," *IEEE Trans. Softw. Eng.*, vol. 5, no. 3, pp. 237–247, 1979.

[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04, 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–150.

[6] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan, "Interpreting the data: Parallel analysis with sawzall," *Scientific Programming*, vol. 13, no. 4, pp. 277–298, 2005.

[7] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *Neural Information Processing Systems (NIPS)*, 2006, pp. 281–288.

[8] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, 2004.

[9] P. L'Ecuyer, "Maximally equidistributed combined tausworthe generators," *Mathematics of Computation*, vol. 65, no. 213, pp. 203–213, 1996.

[10] Celoxica, 2006. [Online]. Available: http://www.celoxica.com

[11] NVIDIA Corporation, *GPU Programming Guide*, 2006.

[12] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

[13] J. Hull, *Options, futures and other derivatives*, 5th ed. Prentice-Hall, 2002.

[14] G. Morris and M. Aubury, "Design space exploration of the European option benchmark using Hyperstreams," in *Proc. International Conference on Field-Programmable Logic and its Applications*. IEEE, 2007.

[15] W. Press, B. Flannery, S. Teukolsky, and W. Vetterling, *Numerical recipes*. Cambridge University Press, 1986.

[16] K. H. Tsoi, K. H. Lee, and P. H. W. Leong, "A massively parallel RC4 key search engine," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2002, pp. 13–21.

[17] K. H. Tsoi, C. H. Ho, H. C. Yeung, and P. H. W. Leong, "An arithmetic library and its application to the n-body problem," in *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2004, pp. 68–78.