Exploring Hybrid-Core Computing for Option Pricing Applications

Brahim Betkaoui, David B. Thomas, Wayne Luk

Department of Computing
Imperial College London
London, England
{bb105,dt10,wl}@ic.ac.uk

ABSTRACT

Hybrid-core computing aims to integrate specialised, reconfigurable hardware with commodity processors in a single architectural model. This paper explores the efficiency of using Hybrid-core computing to accelerate implementations of two financial option pricing applications. We evaluate the performance of the C-coded hybrid-core implementations, and compare it against the performance of existing commodity processors, GPU, and FPGA implementations. The results of this paper demonstrate that Convey's hybrid-core computing achieves around 5 times performance improvement over optimized software, and that hybrid-core implementation is only three times slower than a Tesla 1.3GHz C1060 GPU, and about one order of magnitude slower than a Virtex-5 FPGA implementation. These performance results were obtained with programming and deployment efforts comparable to those of an x86 computer system.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: heterogeneous (hybrid) systems

Keywords

Hybrid-Core computing, Option pricing, FPGA, GPU

1. INTRODUCTION

In the last few years, the cost of commodity processors made them the main choice for high performance computing (HPC). However, HPC systems that are solely based on conventional microprocessors are now facing the problem of flattening clock rates in commodity processors, which has led to a decline in the performance improvement of these processors. Increasing the number of commodity processor within an HPC server may increase the system performance, at the expense of increased programming complexity and ever-increasing costs of power consumption, cooling, and rack space.

One way of addressing this problem is to use a heterogeneous computer system, where commodity processors are augmented with specialized hardware that can accelerate specific kernels. Examples of specialized hardware include graphics processing units (GPUs) and field programmable gate arrays (FPGAs). Such hardware accelerators can offer higher performance and power efficiency compared to commodity processors, but the problem of heterogeneous systems lie in the distribution and management of execution across multiple architectural models. In addition, programming FPGAs requires software developers in HPC to learn a whole new set of skills and hardware design concepts. The required new set of skills remains a large barrier to using FPGAs in HPC.

Hybrid-core computing is another solution, that achieves a compromise between application-specific hardware and architectural integration. An example of such an approach is the Convey Hybrid-Core architecture, where a commodity processor is augmented with an FPGA-based coprocessor. A key feature of this architecture is the programming model, which supports multiple instruction sets in a single address space. The commodity processor executes normal instructions while the coprocessor executes the application-specific instructions. In this paper, we explore the performance of the Convey Hybrid-core server using Convey design application-specific instruction sets. Our key contributions are:

- Application development of two option pricing applications targeting a Convey hybrid-core system.
- Evaluation of its performance showing that about an order of magnitude of performance improvements can be obtained with the programming and deployment ease of an x86 computer system.
- Comparison of the performance results to those of GPU and FPGA implementations.

2. BACKGROUND

2.1 Convey Hybrid-core Architecture

In Convey's hybrid-core architecture, the commodity instruction set, Intel's x86-64, is augmented with application-specific instruction sets (referred to as personalities), which are implemented on a hardware-based coprocessor to accelerate HPC applications. A key feature of this architecture is the support of multiple instruction sets in a single address space. The Convey hybrid-core HC-1 server integrates a commodity host processor, Intel's Xeon processor, with an FPGA-based coprocessor that can be reconfigured dynamically[2]. The Convey Hybrid-core has access to two pools

This work was presented in part at the first international workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2010), Tsukuba, Ibaraki, Japan, June 1, 2010.

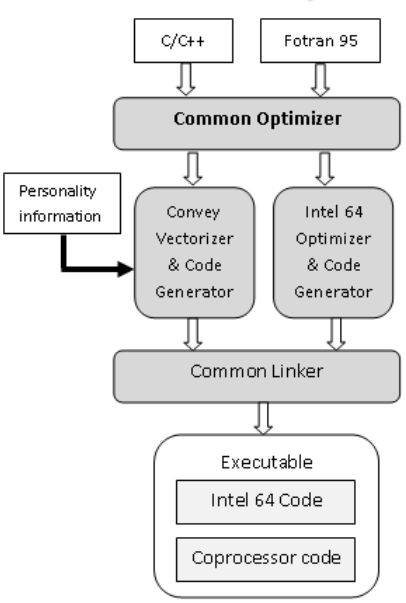

Figure 1: Programming model of the Convey Hybrid-core

of physical memory: the host memory pool, up to 128GB of physical memory located on the x86 motherboard , and the coprocessor memory pool, up to 128GB of standard or scatter-gather DIMM, located on the coprocessor board.

The coprocessor consists of three main components: the Application Engine Hub (AEH), the Memory Controllers (MCs), and the Application Engines (AEs). The AEH represents the central hub for the coprocessor. It implements the interface to the host processor, an instruction fetch/decode unit, and a scalar processor to execute scalar instructions. All extended instructions are passed to the application engines. The coprocessor has eight memory controllers that support a total of 16 DDR2 memory channels offering an aggregate of over 80GB/s of bandwidth to ECC protected memory. Finally, the AEs implement the extended instructions that would be responsible for any performance gains. Each Application Engine is connected to all the eight memory controllers via a network of point-to-point links to achieve a very high sustained bandwidth.

2.2 Convey Hybrid-Core Programming Model

The Convey programming model is shown in Figure 1. A number of key features distinguish the Convey programming model from other coprocessor attached models [3]. Applications are coded in standard C, C++, and Fortran. The user may need to add directives and pragmas to improve the performance though. In an existing application, a routine or a code section may be compiled to run on the host processor, the coprocessor, or both. A unified compiler is used to generate both x86 and coprocessor instructions that are integrated into a single executable, which can run on x86 nodes or on Convey Hybrid-Core nodes. The coprocessor extended instruction set will be defined by the currently loaded personality.

Porting an application to take advantage of the Convey coprocessor capabilities can be achieved by one or more of the following approaches:

- Convey Mathematical Library (CML)
- Compiling one or more routines with Convey's compiler. This can be performed by using the Convey auto-vectorization tool to automatically select and vectorize DO/FOR loops for execution on the coprocessor. Directives and pragmas can also be inserted in the source code to explicitly indicate which part of a routine should execute on the coprocessor. Another way of generating coprocessor code is to write coprocessor assembly language routines that can be called from C, C++, or Fortran.
- Develop a custom personality using Convey's Personality Development Kit (PDK) .

2.3 Convey Personalities

A personality groups together a set of instructions that are geared towards a specific application or a class of applications such as finance analytics, bioinformatics, signal processing, etc. Personalities are stored as loadable bit files on a Convey HC-1 server, and the server can dynamically switch between personalities to execute different algorithms. Each personality consists of two sets of instructions: a scalar instruction set common to all personalities, and an extended instruction set that is application-specific. Convey already provides a number of personalities optimized for certain types of algorithms such as the floating-point vector personalities, and the financial analytics personality.

As an example, the vector personality implements a set vector operations of both integer and floating-point data types. The vector personality is based on a load-store architecture with modern latency-hiding features. There are two variations of this personality: single-precision and double-precision. Figure 2 shows a diagram of one of the 32 function pipes that available in the single precision personality. The double precision version has only two Fuse Multiply-Add (FMA) units and one adder unit.

As another example, the financial analytics personality supports the same features as the double precision vector personality. In addition, it has intrinsic support of functions that are commonly used in finance analytics such as random number generation and math intrinsics.

3. APPLICATION DESIGN

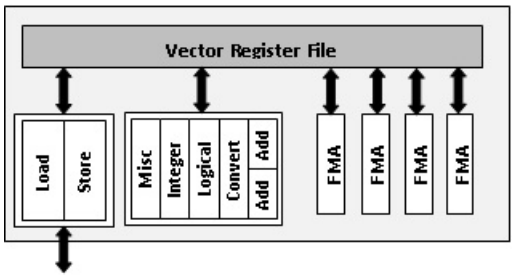

Figure 2: A diagram of 1-of-32 function pipes in the Convey single precision personality

In finance, an option is a financial instrument that is used as a contract between a buyer and seller, in which the buyer has the right, but not the obligation, to exercise the option before its expiration date. Estimating the value of an option is crucial for both buyer and the seller, since the option issuer has to fulfill the contract if the buyer is to exercise the option that he paid for. In this paper, we look at two types of options: Asian options and European options. A detailed description of option pricing can be found in [4] and [1]. We use two different methods to compute the value of European options and Asian options. Numerical integration is used to compute the value of a European option[5], whereas Asian option computation is based on Monte-Carlo methods [6].

Our application benchmarks are all coded in C or C++. We use the latest Intel's Compiler to take full advantage of the advanced optimisation features (automatic processor dispatch, vectorization, and loop unrolling), and multi-threading capabilities supported by Intel's processors. We also use, where applicable, Intel's Mathematics Kernels Library (MKL) which contains highly optimised routines. Multi-threading is used where extra design effort is kept minimal with no significant increase in code complexity.

A typical Convey co-processing flow involve the following[3]:

1. Allocate memory for the data to be processed by the coprocessor on the coprocessor memory.If the data is in the host processor's memory then the data should be migrated to the coprocessor before starting any processing of the data.

2. Instruct the coprocessor to start processing. This can be achieved by either inserting pragmas in the source code to explicitly indicate coprocessor code regions. Another way would be to compile an entire routine for execution on the coprocessor.

3. Wait for the coprocessor to finish processing.

4. Copy the results or data back to the host processor memory if this data is to be accessed/processed by the host processor.

In the process of porting our application to the Convey coprocessor, we performed the following techniques that resulted in the final source code:

- *Data restructuring:* using arrays instead of scalar variables to replace sequential code with parallel code that can be vectorized (should mention the large bandwidth and abundant memory resources available to the coprocessor)

- *Loop interchange:* a loop nest will not vectorize if a loop carried dependency is present in the inner loop. Interchanging the inner and outer loop results in a vectorizable loop nest.

- *Loop unrolling :* in the case of loops with small bodies, unrolling the loop can improve the performance since the amount of computation and the instruction scheduling overlap opportunities are increased, while the loop overhead is amortised [Convey prog_guide]. Unrolling directive or pragmas can be inserted just before the candidate loop for unrolling.

- *Code restructuring:* it is necessary sometime to restructure the source code in order to achieve vectorization. For example, conditional statements within a loop will inhibit the vectorization of that loop.

- *Code inlining:* replacing a function call with its body may be necessary for vectorization. To illustrate this point, we use the code example provided below, where the function call would inhibit the vectorization of the FOR loop.

```
float function(float x, float y)
    return x*y;
}
// before inlining = NO VECTORIZATION!
float x[N], y[N];
float result;
for (i=0; i<N; i++)
    result += function( x[i], y[i]);

// after inlining = VECTORIZATION !
for (i=0; i<N; i++)
    result += x[i]*y[i];
```

In the reminder of this section, we discuss in detail the implementation of two benchmarks on both a commodity processor (Intel Xeon dual-core), and a hybrid-core (Convey HC-1 server). We go through some of the code to demonstrate the design effort required for porting existing software C code to the Convey coprocessor.

3.1 Asian Option Pricing

We use the vdRndGaussian() routine from the Intel MKL to generate the random paths necessary for the Monte-Carlo simulations. The original code of a single-threaded CPU version of the Asian pricing option application is shown in Listing 1.

Listing 1: Asian option pricing: CPU implementation

```
for (i=0; i<mc_cnt; i++){
    error=vdRngGaussian(METHOD, stream,
                  step, &W[0], 0.0, 1.0);
    CheckVslError(error);

    path_sum = S0;
    S = S0;
    for (j=0; j<step; j++){
        S = S*exp(drift + vsqrdt*W[j]);
        path_sum += S;
    }
    mc_call+=fmax(0,path_sum*inv_step-K);
}
```

For each iteration of the outer for loop, a price path is simulated over the lifetime of the option (the inner for loop). The final result is calculated as the average payoff across all these price paths.

Since the CPU target is a multi-core processor, and since the asian option pricing algorithm has inherent data-level parallelism (the path of sums can be computed independently) , we would certainly benefit from having more than one thread running the benchmark. Each thread will be

working on different data items in a bid to improve the performance of the application. The number of threads should be at least equal to the number of cores present on the CPU target. In our case, we have a dual-core processor, and the data can be easily divided between multiple threads, so we use two threads each performing half the calculations. We can then add up the results of each thread to obtain the option value.

The original code portion show in Listing 1 was ported for Convey coprocessor acceleration and the resulting code is shown in Listing 2. The Finance Analytics personality is used as it supports vector instructions that can generate Mersenne Twister random numbers very efficiently[3].

Listing 2: Asian option pricing Convey HC-1 implementation

```
S=cny_cp_malloc(mc_cnt*sizeof(double));
path_sum=cny_cp_malloc(mc_cnt*sizeof(double));

#pragma cny begin_coproc

//initialization
for (i=0; i<mc_cnt; i++){
        S[i] = S0;
        path_sum[i] = S0;
}

// Simulation paths
for (j=0; j<step; j++){
#pragma cny unroll(4)
  for (i=0; i<mc_cnt; i++){
     S[i] = S[i]*exp(drift
            + vsqrdt*mersenne_tbl());
     path_sum[i] += S[i];
  }
}

// average payoff
for (i=0; i<mc_cnt; i++)
  mc_call+=fmax(0,path_sum[i]*inv_step-K);

#pragma cny end_coproc
```

We first allocate the memory space for the total number of price paths and the asset prices in the coprocessor memory. Then we initialise the allocated arrays S and path_sum with the initial option price.

In the path simulation loop, the price paths are computed in parallel since they are independent from each other.This is achieved by vectorizing of the innermost loop. For the random paths, we use the Mersenne-Twister random number generator routine provided in the Convey finance analytics personality.

Finally, the average payoff across all these paths is computed as the expected value of the Asian call option at time step-1. This value is then passed to the code executing on the host processor to calculate the arithmetic call option value at the present time

3.2 European Option

We compute the price of a European option by evaluating the following integral

$$V(x,t) = A(x)\int_{-\infty}^{\infty} B(x,y)V(y,t+\triangle t)dy \qquad (1)$$

where $V(x,t)$ is price of the option, x is the value of the underlying asset, t denotes time. The full derivation of equation 1 is provided in [1]. The integral is implemented with a FOR loop that iterate through the number of integration grid points N as shown in the code snippet of Listing 3.

Listing 3: European option pricing: CPU implementation

```
float evaluate(float V, float y, float x,

    float halfk, float C, float factor){
    float Bxy = exp(halfk*y-pow(x-y,2)/C);
    float f = Bxy * V;
    return factor*f;
}

float EvaluationCore(int start, int N,
                IntegralData* point){
  float integral=0;
  for (int i = start; i < N; i++){
    integral += evaluate(point.V[i],
                         point.y[i],
                         point.x[i],
                         point.halfk[i],
                         point.C[i],
                         point.factor[i]);
  }
  return integral;
}
```

We can easily see that this code will benefit from multithreading considering that the computations carried out inside the FOR loop are independent. We use two threads each performing half the calculations and then sum up the resulting partial integral values to obtain the value of the European option.

There were no significant changes made to the original code to port it to the Convey coprocessor. The only thing that was inhibiting the vectorization of the FOR loop in the EvaluationCore routine was the function call to the Evaluate routine. We worked around this problem by inlining the Evaluate routine inside the EvaluationCore function as shown in Listing 4. Further optimizations included inserting a pragma to manually unroll the FOR loop in the Evaluation Core routine.

Listing 4: European option pricing: Convey HC-1 implementation

```
#pragma cny dual_target(1)
double EvaluationCore(int start, int count,
       double* V, double* x, double* y,
       double* half_k, double* C, double* factor)
{
  double integral=0;

#pragma cny unroll(4)

  for (unsigned i = start; i < N; i++) {
       integral+=factor[i]*(exp((half_k[i]*y[i]
      -((x[i]-y[i])*(x[i]-y[i]))/C[i]))*V[i]);
    }
   return integral;
}
```

4. EVALUATION AND COMPARISON

In this section, we evaluate the performance of our benchmarks on the Convey HC-1 server. A performance comparison is then established with two CPU implementations,

Table 1: Execution Times

Implementation	Asian option		European option	
	Single	Double	Single	Double
CPU single-threaded	8,333 ms	14,727 ms	25 ms	70 ms
CPU multi-threaded	4,446 ms	8,378 ms	19 ms	40 ms
Convey HC-1	1,641 ms	1,641 ms	14 ms	7 ms
GPU	440 ms	-	3 ms	5 ms
FPGA	115 ms	-	6 ms	21 ms

Table 2: Comparison VS. Software speedup

Implementation	Asian option		European option	
	Single	Double	Single	Double
CPU single-threaded	0.53x	0.57x	0.76x	0.57x
CPU multi-threaded	1x	1x	1x	1x
Convey HC-1	2.7x	5.1x	1.4x	5.6x
GPU	10x	-	6.3x	7.5x
FPGA	38.6x	-	3.4x	1.9x

Table 3: Comparison VS. Convey HC-1 speedup

Implementation	Asian option		European option	
	Single	Double	Single	Double
Convey HC-1	1x	1x	1x	1x
GPU	3.7x	-	2.33x	1.4x
FPGA	14.7x	-	1.16x	0.33x

single and multi-threaded, a GPU version, and an FPGA implementation.

For the Asian option pricing benchmark, we use the same parameters as in [6], i.e. 1,000,000 simulations over a time period of 356 steps. Similarly for the European option pricing application, the parameters are selected based on the work in [5].

The Convey HC-1 has an Intel Xeon Dual-core 5138 running at 2.13GHz with 8GB of memory, and a Coprocessor that consists mainly of four Xilinx V5LX330 FPGAs running at about 300MHz, and that have access to 16GB of memory. The Convey host processor was used to run the reference CPU benchmarks. An nVidia Tesla 1.3GHz C1060, and a Xilinx Virtex-4 xc4vlx160 (running at 100MHz) is used in [5] for the European option pricing benchmark. For the Asian option pricing application, the author in [6] used the Tesla 1.3GHz C1060 GPU, and a Virtex-5 xc5vlx330t FPGA clocked to 200MHz, for the GPU and FPGA implementations respectively.

4.1 Performance comparison

Table 1, 2, and 3 show the performance of Convey HC-1 implementation compared to the performance of the CPU, GPU, and FPGA implementations.

These results show that using the Convey coprocessor yields a performance improvement of 2-5 times over an optimized multi-threaded software implementation running on a CPU with two cores. The major reason for this performance improvement is the vectorization of the FOR loops that were deemed to be the bottleneck in the option pricing benchmarks. The single-precision personality has 32 function pipes implemented on 4 FPGAs, and each function pipe has four Fused Multiply-Add (FMA) units and four adder unit. So a total of 128 pipelined FMA units and 128 adder units are used, in addition other miscellaneous and integer units . This number is halved in the double-precision vector and financial analytics personalities. The finance analytics implements some common math functions such exponential and natural logarithm intrinsically. It can also generate very efficiently Mersenne-Twister random numbers that are used in our Asian option pricing benchmark.

For the Asian option pricing application, the finance analytics does not support single-precision floating-point vector instructions. So the single-precision version of the CPU, FPGA, and GPU are compared to a double-precision Convey implementation. The GPU version is about 4 times faster, while the FPGA version is about 15 times faster. We would expect these values to decrease if we were to implement a double precision version of either the FPGA implementation or the GPU implementation. We would also predict that a single-precision version of the finance analytics will outperform its double-precision counterpart.

As for European option pricing benchmark, the convey HC-1 is 5 times faster than the multi-threaded CPU implementation. This performance gain is only about 40% in the case of single-precision floating-point implementation. This was an unexpected result considering that the single-precision personality has twice the amount of FMA and adder units compared to the double-precision personality. Using the Simulation Performance Analysis Tool (SPAT) provided by Convey, revealed that only about 25-50% of the total FMA units were used in the single-precision personality. Convey Customer support confirm that while the exponential function, *exp()*, is implemented by a Taylor series algorithm, the double-precision version of the function has a better utilization of the FMA units due to function inlining. The single-precision version of this function is not yet inlined, and thus needs more tuning. We also notice that the GPU implementation is only about 2 times faster than the Convey implementation; while the double precision FPGA implementation is outperformed by the Convey Hybrid-core by a factor of 3. The FPGA design [5] involves Hyperstreams and Handel-C which results in less efficient designs compared to hand-coded HDL designs. This may be the reason the FPGA implementation is exhibiting such modest performance, in particular for the double precision version.

4.2 Performance gain breakdown

Table 4 shows a list of different programming techniques that are used to port the original C/C++ code to the Convey coprocessor and the effect of such techniques on the performance. In some cases, it is worth noting that not all techniques can be applied to the two benchmarks. We used the hyphen sign '-' to indicate that the technique does not

Table 4: Some Typical techniques to improve the performance of the Convey coprocessor code

Technique	**Asian option**	**European option**
Loop interchange	2.4x	-
Loop unrolling	1.22x	1.15x
Code restructuring	1.74x	-
Code inlining	-	4.86x
Total	5.1x	5.6x

result in vectorizable code or any performance gain. In the Asian option benchmark, code restructuring consisted of replacing a 'conditional statement' within a FOR loop, with the intrinsically supported routine 'fmax'.

The performance gains achieved in the Asian option pricing application, are mainly due to loop interchange and code restructuring. These two techniques are employed to generate vectorized coprocessor code that is necessary if we are to obtain any performance gains on the Convey coprocessor. Failing to achieve vectorization leads to scalar coprocessor code which runs significantly slower than a commodity processor. Similarly, code inlining is responsible for code vectorization, and hence, the main contributor to the performance gain in the European option pricing application. Loop unrolling does not generate vectorized code, and is rather used to optimise the already vectorized coprocessor code.

5. DISCUSSION

One of the goals of Convey Hybrid-Core architecture is to allow software developers and scientists to take advantage of the benefits of reconfigurable hardware without any hardware design skills. They can accelerate their applications on the Convey coprocessor by using an application-specific personality. Using the Convey HC-1 server, we achieved decent performance improvements for our benchmarks, using only the C programming language assisted by few Convey pragmas to help the compiler generate efficient coprocessor code. Assuming some familiarity with the Convey programming model, we estimate the design effort to be very close and comparable to that of optimizing software for commodity processors.

6. CONCLUSION

This paper explores the acceleration of two finance analytics applications using application-specific instructions, that are integrated with a commodity instruction set in a Hybrid-Core computer system. The application-specific instructions are provided by Convey Computer^{TM}and are geared towards financial analytics applications and floating-point vector operations. The benchmarks applications were coded in C, and then optimized using software-like optimization techniques such as loop unrolling, and function inlining.

The performance results of the Convey hybrid-core server are compared with multi-threaded CPU, GPU, and FPGA implementations. The Convey hybrid-core implementation is about five times faster than an optimized multi-threaded CPU implementation. The performance of the hybrid-core was only about three times slower than the Tesla 1.3GHZ C1060 GPU implementation, and only about an order of magnitude slower than an HDL-coded Virtex-5 FPGA implementation. However, the efforts of porting existing application code to the Convey hybrid-core server are estimated to be comparable to those of optimising a software implementation on an x86 commodity processor. We expect the performance results to improve in the future, with advances in compiler technology for hybrid-core systems.

Future work includes exploring the development and performance of user-defined custom personalities. We aim to exploit the high-performance memory subsystem designed by Convey to implement algorithms with high memory bandwidth requirements. We hope to be able provide a more accurate representation of the advantages and disadvantages of using Convey hybrid-core architecture in terms of performance, power consumption, and development efforts.

7. ACKNOWLEDGMENTS

The support of Imperial College London Research Excellence Award, UK Engineering and Physical Sciences Research Council, Convey Computer Corporation, and Xilinx, Inc. is gratefully acknowledged.

8. REFERENCES

[1] F. Black and M. S. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–54, May-June 1973.

[2] Convey Computer Corporation. *Convey Reference Manual*, September 2009.

[3] Convey Computer Corporation. *Convey Programmers Guide*, January 2010.

[4] E. Sueli and D. F. Mayers. *An Introduction to Numerical Analysis*. Cambridge University Press, 2006.

[5] A. H. Tse, D. B. Thomas, and W. Luk. Accelerating quadrature methods for option valuation. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:29–36, 2009.

[6] A. H. Tse, D. B. Thomas, K. H. Tsoi, and W. Luk. Efficient reconfigurable design for pricing Asian options. *This volume*.