Abstract. Heterogeneous computing platforms with multicore host system and many-core accelerator devices have taken a major step forward in the mainstream HPC computing market this year with the announcement of HP Apollo 6000 System’s ProLiant XL250a server with support for Intel® Xeon Phi™ coprocessors. Although many application developers attempt to use it in the same way as GPGPU acceleration platforms, doing so forfeits the processing capability of multicore host processors and introduces power inefficiency in corporate IT operations. In this paper, we propose an application optimization framework to turn a sequential legacy application into a highly parallel application to make use of the hardware resources both on the host CPU and on the accelerator devices to enable simultaneous heterogeneous computing. As a case study, we look at how to apply this framework and adopt a structured methodology to adapt a European option pricing application to take advantages of a heterogeneous computing environment.

1 Introduction

Since the Intel® Xeon Phi™ coprocessor was launched three years ago, many papers have been written about its performance in comparison with GPGPUs. Relatively few papers have been written on the comparison of available programming models and methodologies. This is perhaps partly because of the maturity of development tools and the environment, or partly because of hardware availability. With the release of Intel Parallel Composer XE 2015 and HP’s release of the Apollo 6000 ProLiant XL250a server early this year, it’s now time to look at how developers can follow a step-by-step methodology in a structured framework to develop highly parallel applications that can run effectively in an integrated heterogeneous environment.
1.1 HP Apollo 6000 as a Heterogeneous Computing Platform

The HP Apollo 6000 System with ProLiant XL250a servers is a practical HPC platform to enable heterogeneous computing in a compact form factor. The Apollo a6000 chassis is a 5U enclosure with shared power, cooling, and I/O infrastructure with ten independent server trays. Each HP ProLiant XL250a server consumes two slots, supporting two Intel Xeon E5-2600 v3 processors, two Intel Xeon Phi coprocessors, 16 DIMMs, six SFF disk drives, and many networking options. Chassis power is supplied by a 1.5U HP Apollo 6000 Power Shelf, each of which supports up to six Apollo a6000 chassis. Supported E5-2600v3 series processors can have up to 16 cores and 32 threads running AVX code at up to 2.5 GHz. When configured with Intel® Xeon Phi™ Coprocessors, each server is capable of 6.7 to 7.7 peak Single Precision Tera FLOPs.

1.2 Intel Parallel Composer XE 2015

The development of application software on heterogeneous systems involving multicore and many-core product such as the HP ProLiant x1250a is made possible by Intel Parallel Composer XE 2015. It includes optimizing compilers, performance libraries and an application profiler. On the heterogeneous system, two sets of tools are installed. Compilers and libraries are provided for the host CPU and libraries for the accelerator devices. The developer can choose to develop either native or offload application to run on the many-core accelerator device.

A native application is one in which the binary resides on the accelerator device. To develop a native application, you just have to invoke the compiler with switches designating the coprocessor’s architecture. An offload application is initiated on the host processor but a portion of the code executes on the many-core devices. To build an offload application, the programmer has to mark the code section to be run on the many-core devices. The compiler will first compile the whole program as if it runs entirely on the host processor and then compile the section marked to run on the many-core. The two different executable files co-reside in a single compiler output file. The compiler runtime checks the availability of the coprocessor device, and if it’s available, the many-core executable file will be sent to the device for execution.

2 Step-wise Performance Optimization Framework

The Step-wise Optimization Framework is a collection of methodologies and tools that enables the developers to express parallelism for multicore and many-core platforms. The main objective of the step-wise performance optimization framework is to turn the legacy, un-optimized, sequential program into a scalable, vectorized parallel application that executes simultaneously on the multicore processor (Xeon) and on the many-core processor (Xeon Phi) in a heterogeneous computing environment.
This model walks the developer through the key steps essential to extracting the maximum application performance from multilevel parallelism. The exact order of these steps is not that important, but the unique issues and challenges at each step have to be addressed.

The purpose of having a framework is to create a high level abstraction model of performance optimization and to isolate and address source code related issues unique to each step and quantify the performance gains at each step.

2.1 Leverage Optimized Tools

For the legacy software written several decades ago and requiring a serious modernization effort, the first step is to find out if the choice of tools and library is still valid in today’s environment. The choice made in this step is going to affect the whole code modernization project. The criteria in choosing tools and libraries for a project obviously depend on the project objective. For most performance optimization, a good choice would be the one that makes it easiest for you to extract parallelism at various levels of the underlying hardware. The source code written in Turbo C, which might have been a good tool 30 years ago on 486DX processors, may have to be ported to use Intel’s compiler for Intel Xeon E5 2600 v3 processors.

Choosing the right tools and libraries also allows you to leverage existing solved problems and concentrate on the problem at hand. For example, 60% of quantitative finance problems are using the Monte Carlo method in which a random number generator is involved. Instead of optimizing every random number generator in every Monte Carlo problem, we can use the random number generator libraries. This gives you more energy and undivided attention to focus on the problem that no one has solved before.

2.2 Scalar and Serial Optimization

Scalar and Serial Optimization eliminates the redundancy in your original algorithm implementation. Places where floating point operations take places certainly fall into this category. Your legacy code developed for 486DX processors might be using x87 floating point unit, whereas your target for your code project might be core i7 using a vector processing unit. An unsuspecting compile switch may give you the wrong target for FP computation.

Optimization at this level may include language related optimizations as well. For example, automatic C/C++ mandated type conversion is another source of concern at
this step. Type conversion takes time, and even when required is auxiliary to your problem at hand. For example, when you see float converted to double only to be converted back again, you need to investigate if it is possible to keep data in single precision all the time.

Other language related changes make it easier for the compiler to work with the code using the features in the next few steps of the framework. For example, self-modifying code makes parallelization close to impossible, as do loops with varying upper or lower bounds.

2.3 Vector Programming

Vector programming is the parallelism offered by the vector processing units, which are part of the instruction set architecture, or ISA, in modern processor design. This level of parallelism is also called SIMD, or sometime less accurately, data parallelism. The programmer takes advantage of the contiguous layout of the data so that a vector of N-elements can be processed at the same time. The biggest hurdle to efficient vector processing is the break of execution flow, either via branch statement or function calls. This is simply because the synchronous SIMD model does not allow execution branches to multiple places. With vector compare, MASK and blend instructions, branch statements can be serialized, then combined back into vectors processing. For the function call, the compiler provides a mechanism for the programmer to create a new function with vector parameters that allows the execution to jump into the new location to resume vector execution.

Successful vector programming depends on the data layout in the memory cache as well. Laying out data of elementary C types contiguously in memory has a significant benefit because the hardware has the capability to anticipate and prefetch the data and pack them into vector registers. If the data are aggregations of many different types, laying them out in arrays of structures, or AOS, has big advantages over structures of arrays.

2.4 Thread Parallelism

Thread parallelism leverages the CPU cores and hardware threads inside the multi-core or many-core processors. The foundation of the parallelism is built on the repetitive hardware resources, and the assumption is that these CPU cores and threads can function independently. For the most part, this is true. If we can split the total workload into M-parts for M threads and keep the connections between them to a minimum, then each thread indeed makes the most contribution to the total workload.

Thread parallelism depends on the memory bandwidth. As each thread is executing vector instructions, the aggregate memory access for all the cores in a processor is tremendous. Because the speed and capacity of the memory subsystem is unable to satisfy the demand from the processor, the processor cores have to rely on a hierarchical cache system to hide the memory access latency.
In modern processors, multiple interconnect hardware such as memory controllers and processor cores also create topology challenges, which makes some memory remote and some memory local to a processor. Making the memory a processor cares about local and all the rest remote is a unique optimization problem in this step.

2.5 From Multicore Computing to Heterogeneous Computing

For large scientific applications, we still cannot get the results quickly enough even if we distribute the work among all the cores in a server and use vector instructions. For these problems, heterogeneous computing delivers a better solution by combining the benefit of both multicore and many-core computing.

In our framework, we have to scale the multicore solution to the many-core platform first, and combine the multicore and many-core solution in an integrated heterogeneous solution. From a high level architectural aspect, the many-core processors and multicore processors are very similar in that both include copious processing cores with vector processing units. There are some micro architecture-level differences, such as the number of vector lanes, the number of cores and their interconnect topology, the memory bandwidth interface, the processing frequency, and some instructions related to further extracting performance due to these differences. With the Intel Compiler, the programmer can quickly extend the parallelism of a program from multicore to many-core by using minimal porting effort specific to the many-core micro architecture. The programmer can choose to build a native or an offload solution as we mentioned in the previous section.

Combining both multicore and many-core solutions in an integrated heterogeneous solution is the center of the challenge in this step. In a heterogeneous system such as the HP ProLiant XL250a server, the host multicore processors and accelerator cards based on many-core processors do not share any physical memory or logical address space. The program and data for the many-core system originally reside on the host system. To utilize the many-core accelerator card, program and data have to be copied from the host system to the accelerator card, and the resulting data have to be transferred back from the card to the host. Heterogeneous processing will break a big workload into two portions, one processed by the host multicore processors and the other processed by the accelerator cards. Simultaneous heterogeneous computing achieves the maximum performance by utilizing concurrent processing on both multicore host processors and many-core processors.

3 Case Study – Financial Derivative Pricing

We applied our optimization framework to a unique and common set of problems in the banking industry known as financial option pricing algorithms.

The first algorithm is Black-Scholes, which is an implementation of the Black-Scholes-Merton Formula, a closed-form solution to the Black-Scholes equation.
Monte Carlo uses a statistical computing approach to model the uncertainties as random numbers obeying certain laws of distribution, and calculates the derivative based on the average pay off. Based on the fact that a random number can be used in a different unrelated payoff or not, we have two versions of Monte Carlo algorithms.

The third algorithm is a binomial option pricing algorithm. The algorithm models the stock price changes as a series of ups and downs leading from the current time to the expiry. The pricing process goes from the expiry to the current time using backward sweep.

The fourth algorithm is Barone-Adsi and Whaley, or BAW, an approximation of American options using the Newton-Raphson method for non-linear equations.

As an exercise, we only collected the performance numbers at the end of the vectorization and parallelization step for single precision and double precision. We found the data can challenge conventional thinking. Our experience is that vector programming delivers performance gains multiple times higher than from the vector lanes alone. This is due to new instructions available in the vector ISA but not available in the scalar ISA, and means that vector programming has the side-effect of using a better instruction mix for your application.

Table 1. Performance Gain from Step 3 and Step 4 using Step-wise Optimization Framework

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Vector Programming (SP, DP)</th>
<th>Multithreaded Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes</td>
<td>9.129X</td>
<td>8.251X</td>
</tr>
<tr>
<td>Monte Carlo (Pre)</td>
<td>13.622X</td>
<td>7.264</td>
</tr>
<tr>
<td>Binomial</td>
<td>6.803X</td>
<td>3.835X</td>
</tr>
<tr>
<td>BAW</td>
<td>7.421X</td>
<td>-</td>
</tr>
</tbody>
</table>

In the rest of this section, we move code for one of these algorithms from multicore to heterogeneous computing as we introduce the simultaneous multicore and many-core environment. In general, we found that we are able to achieve high performance by using the programming techniques in the framework.

3.1 Monte Carlo using MKL Random Number Generator

Monte Carlo is a numerical method that uses statistical sampling techniques to approximate solutions to quantitative problems. The contemporary version of the Monte Carlo algorithm was first used by Stanislaw Ulam, while he was working on the Manhattan project in the mid-1940s. Nicholas Metropolis was the first to make the connection between the casino and the algorithm and coined the term Monte Carlo. Since then, Monte Carlo has been widely used in engineering, physics, molecular dynamics, and in calculating integrals with complicated boundary conditions.

In 1973, Fisher Black and Myron Scholes published their historical paper and introduced what was later known as the Black-Scholes Option Pricing Model for financial derivatives. As the rest of the world was still trying to make sense of the Black-Scholes Model, an actuarial professor from the University of British Columbia, Phelim Boyle, introduced the Monte Carlo method to verify the result from the closed form Black-
Scholes-Merton Formula. In his article, he takes the example of a European call option and calculates its price using the Monte Carlo method. We are going to use the same algorithm for our heterogeneous computing investigation.

The advantage of using a Monte Carlo simulation is that it can be used when the payoff depends on the path followed by the underlying variable S leading toward the expiration and the situation when payoffs take place multiple times during the life of the option. It is particularly useful when the payoff function involves multiple independent variables. When all other analytical methods fail, Monte Carlo then becomes the only choice.

3.2 Algorithm Implementation

A Monte Carlo simulation uses the risk-neutral valuation method to value an option. It samples a path to obtain the expected payoff in a risk-neutral world and then discounts the payoff to current value using a risk-free interest rate. Let’s consider a stock option for a stock with current price S that provides a payoff at time T. Assuming the interest is constant, we can value the derivative as follow:

- Sample a random path for S in a risk-neutral world.
- Calculate the payoff from the derivative.
- Repeat previous two steps to get many sample values of the payoff from the derivative in a risk-neutral world.
- Calculate the mean of the sample payoffs to get an estimate of the expected payoff in a risk-neutral world.
- Discount the expected payoff at the risk-free rate to get an estimate of the value of the derivative.

It follows from the central limit theorem that to reduce the standard deviation by half, the number of sampling path needs to be quadrupled. In other words, the standard error for Monte Carlo converges at the rate of $O\left(\frac{1}{\sqrt{N}}\right)$.

Original implementation of Monte Carlo European Option Pricing

```c
#include "MonteCarlo.h"
#include <math.h>
#include <tr1/random>
#ifndef max
#define max(a,b) (((a) > (b)) ? (a) ; (b))
#endif
void MonteCarlo(
    float *CallResult,
    float *CallConfidence,
    float *S,
```
float *X,
float *T
)
{
    typedef std::tr1::mt19937                     ENG;
    typedef std::tr1::normal_distribution<float> DIST;
    typedef std::tr1::variate_generator<ENG,DIST> GEN;
    ENG  eng;
    DIST dist(0,1);
    GEN gen(eng,dist);

    for(int opt = 0; opt < OPT_N; opt++)
    {
        float VBySqrtT = VOL * sqrt(T[opt]);
        float MuByT = (RISK - 0.5 * VOL * VOL) * T[opt];
        float Sval = S[opt];
        float Xval = X[opt];
        float val = 0.0f, val2 = 0.0f;

        for(int pos = 0; pos < RAND_N; pos++)
        {
            float callValue = max(0.0, Sval *exp(MuByT + VBySqrtT * gen())) - Xval);
            val  += callValue;
            val2 += callValue * callValue;
        }

        float exprt = exp(-RISK *T[opt]);
        CallResult[opt] = exprt * val / (float)RAND_N;
        float stdDev = sqrt(((float)RAND_N * val2 - val * val)/ ((float)RAND_N * (float)(RAND_N - 1)));
        CallConfidence[opt] = (float)(exprt * 1.96 * stdDev / sqrt((float)RAND_N));
    }
}

3.3 From Un-optimized Scalar Serial Program to Vector Parallel Program

Applying the Step-wise Optimization Framework, we can quickly achieve the performance gain we expect.

Step 1, Using Optimizing Tools and Library. In this step, we addressed two issues: compiler based numerical optimization in code generation, and use of the MKL library. The serial interface of the random number generator has effectively become the first
bottle neck that we needed to remove. By doing both, we improved the performance by 40%.

Step 2, Numerical Optimizations. In this step, we selected a precision for our problem and ensured the entire computation took place in that precision, avoiding any unnecessary data conversions, enforced by the C/C++ auto conversion rule and C constant type misinterpretations were addressed. Transcendental function calls were all ensured to be precision specific. The compiler switch `-fimf-precision-low` allows runtime libraries to select the fast version of runtime math functions, and `--fimf-domain-exclusion=31` avoids unnecessary floating point operation flag checks.

Step 3, Vector Programming Using pragma SIMD. In this step, we applied SIMD parallelism to the inner loop so that each SIMD lane calculates a unique sample from one option. We ensured the random number buffer is aligned at a 64-byte boundary or better. Therefore, the compiler can generate automatic vector load and store instructions. Notice that there are two reduction operations. The compiler has to carry the result across different iterations. `#pragma reduction` tells the compiler to generate reduction code.

Step 4, Core Parallelization. In this step, we have to apply core-parallelism to the outer loop so that each thread calculates a different option simultaneously. Since two threads share each core and its execution resources, we have to ensure that the combined footprint from both threads that share a core fits into that core’s L2 cache. If it doesn’t, i.e. it is too big for the cache, we have to tile the footprint in, one at a time.

Here is the sample source code of the Monte Carlo European options pricing algorithm using Step 1 to Step 4 of the Step-wise Optimization Framework

```c
void MonteCarlo(
    float *CallResult,
    float *CallConfidence,
    float *S,
    float *X,
    float *T
)
{
    #ifdef _OPENMP
        kmp_set_defaults ("KMP_AFFINITY=compact,granularity=fine");
    #else
        int NTHREADS = 1;
    #endif

    #pragma omp parallel
    {
        #ifdef _OPENMP
        int threadID = omp_get_thread_num();
```
```
#else
int threadID = 0;
#endif
unsigned int randseed = RANDSEED + threadID;
__attribute__((aligned(1024))) float random[BLOCKSIZE];
vslNewStream(&Randomstream, VSL_BRNG_MT2203, randseed);
const int nblocks = RAND_N / BLOCKSIZE;
#pragma omp for
for (int opt = 0; opt < OPT_N; opt++)
{
    float VBySqrtT = VLOG2E * sqrt(T[opt]);
    float MuByT = RVVLOG2E * T[opt];
    float Sval = S[opt];
    float Xval = X[opt];
    float val = 0.0f, val2 = 0.0f;
    for (int block = 0; block < nblocks; ++block)
    {
        vsRngGaussian(VSL_RNG_METHOD_GAUSSIAN_ICDF, Randomstream, BLOCKSIZE, random, MuByT, VBySqrtT);
        #pragma vector aligned
        #pragma simd reduction(+:val) reduction(+:val2)
        #pragma unroll(4)
        for (int pos = 0; pos < BLOCKSIZE; pos++)
        {
            float callValue = Sval * exp2f(random[pos]) - Xval;
            callValue = (callValue > 0) ? callValue : 0;
            val += callValue;
            val2 += callValue * callValue;
        }
    }
    const float  exprt = exp2f(-RLOG2E*T[opt]);
    CallResult[opt] = exprt * val * INV_RAND_N;
    const float stdDev = sqrtf((F_RAND_N * val2 - val * val) / STDDEV_DENOM);
    CallConfidence[opt] = exprt * stdDev * CONFIDENCE_DENOM ;
}  // end of for opt
vslDeleteStream(&Randomstream);
}  // end ofomp parallel
}  //end of MonteCarlo

Performance Tables. The performance gains from the first 4 steps can be summarized in Table 2.

<table>
<thead>
<tr>
<th>Step</th>
<th>Before Optimization</th>
<th>After optimization</th>
<th>Gain Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>15.563 opt/sec</td>
<td>359.097 opt/sec</td>
<td>23.07</td>
</tr>
<tr>
<td>Step 2</td>
<td>359.097 opt/sec</td>
<td>581.137 opt/sec</td>
<td>1.62</td>
</tr>
</tbody>
</table>

Table 2. Performance Gain from Step 1 to Step 4 using Step-wise Optimization Framework
3.4 Scale from Multicore to Heterogeneous Computing

Our first step in Step 5 is to ensure that our optimizations we applied on the multicore platform can extend to the many-core platform. We would like to scale the multicore solution to many-core. The first decision we have to make is whether we should use offload or native mode for many-core execution. If the amount of data transfer between host and accelerator card is relatively small, the single entry benefit of offload execution becomes very attractive. It saves the manual transfer of program and data by the program user.

The build process of an offload executable is greatly simplified with one tool’s invocation and one executable file. For applications that have to use performance libraries, such as the Intel Math Kernel Library, an additional switch such as –mkl has to be added. However, underneath, a lot of action takes place. Two compilers are invoked; one compiles the source files to run on the host multicore, and the other compiles the source files to run on many-core coprocessors. If MKL has to be used in both places, –mkl expands the include path to find appropriate .h files at compiler invocation, and the library file path to find appropriate .so files at linker invocation.

**Annotate Offload Function.** Our first task is to annotate a piece of work so that the compiler can generate a code sequence to be carried out on the coprocessors. For that task,

Definition of an offload function.

```c
__declspec(target(mic))
void MonteCarlo(float *CallResult,
float *CallConfidence,
float *S,
float *X,
float *T)
{
   ...
}
```

The Intel Compiler compiles it once for code to be executed on the multicore host processors and compiles it a second time for code to execute on the many-core coprocessors.

**Annotate Offload Global Variables.** We have to treat the global variables the same way we treat the C subroutines, which asks the compiler to reserve the memory space once in the address space of the host multicore system and once again for the many-core coprocessors.
This arrangement is necessary because the multicore host and many-core coprocessors do not share any physical or virtual memory. For variables shared between the two heterogeneous systems, the C runtime library has to ensure that changes to one memory system are automatically propagated to the other system.

**Offload Invocation.** The syntax to offload a function is very similar to that of a function call, except `pragma offload` is used to specify where it is offloaded. There is an additional “in” clause to specify the copy action of input data from host to the coprocessor and an “out” clause to specify the copy back action of the result from the coprocessor to the host.

Here is the sample source code to invoke Monte Carlo European options pricing on the many-core coprocessors:

```c
#pragma offload target(mic:0) in(StockPriceList,
                           OptionStrikeList,
                           OptionYearsList : length(OPT_N))
out(CallResultList,
    CallConfidenceList:length(OPT_N))
MonteCarlo (CallResultList,
            CallConfidenceList,
            StockPriceList,
            OptionStrikeList,
            OptionYearsList);
```

Unless otherwise specified, offloading a function is a synchronous operation. The host multicore executable stalls until the offload function comes back from the coprocessor.

**Asynchronous Offload.** To allow the host multicore system to continue running before the offload program returns, we need to use a conditional variable associated with the offload program. The variable is signaled when the offload function is complete.

Example code defining a conditional variable for asynchronous offload invocation and waiting for the results. Any statement at line 12 will be executed concurrently after offload starts and before offload completes.

```c
0. char sigComplete0;
1. #pragma offload target(mic:0) in(StockPriceList,
2.                                OptionStrikeList,
3.                                OptionYearsList : length(OPT_N))
4. out(CallResultList,
5.      CallConfidenceList:length(OPT_N))
6.   signal(&sigComplete0)
7.   MonteCarlo (CallResultList,
8.                   CallConfidenceList,
9.                   StockPriceList,
10.                  OptionStrikeList,
11.                  OptionYearsList);
```
Asynchronous offload prevents the host from entering into a stall state and allows continued program execution before the offload is completed. At line 12, we can put any additional calculations. That’s the work the host can be doing after the offload is started.

The host processor can also explicitly wait for the completion of offload by checking the conditional variable.

**Simultaneous Heterogeneous Processing** If we give some meaningful work to the host processor while the offload program is busy with the coprocessor, this work can be done simultaneously in a heterogeneous environment.

Example code using asynchronous offload to achieve simultaneous heterogeneous processing:

```c
char sigComplete0;
#pragma offload target(mic:0) \ 
in(OffloadStock, OffloadStrike, OffloadTime : length(OPT_N))
out(CallResultList, OffloadConfidence:length(OPT_N))
signal(&sigComplete0)
MonteCarlo (OffloadResult, OffloadConfidence, OffloadStock, OffloadStrike, OffloadTime);
MonteCarlo (HostCallResult, HostCallConfidence, HostStock, HostStrike, HostTime);
#pragma offload_wait target(mic:0) wait(&sigComplete0)
```

**Simultaneous Heterogeneous Processing with Multiple offload** For heterogeneous systems with multiple many-core coprocessor devices, we can use the asynchronous offload mechanism to achieve simultaneous heterogeneous processing between two devices and also with host processors. The function at line 7 runs on the mic0 coprocessor. Simultaneously, the function at line 18 runs on the mic1 coprocessor, and the function at line 23 runs on the host coprocessors.


```c
char sigComplete0;
#pragma offload target(mic:0) \ in(OffloadStock0, OffloadStrike0, OffloadTime0 : length(OPT_N))
```
4. out(OffloadResult0, OffloadConfidence0: length(OPT_N))
5. signal(&sigComplete0)
6. MonteCarlo (OffloadResult0, OffloadConfidence0, OffloadStock0, OffloadStrike0, OffloadTime0);
7. #pragma offload target(mic:1) in(OffloadStock1, OffloadStrike1, OffloadTime1 : length(OPT_N)) out(offloadResult1, OffloadConfidence1:length(OPT_N)) signal(&sigCompleteS)
8. MonteCarlo (OffloadResult1, OffloadConfidence1, OffloadStock1, OffloadStrike1, OffloadTime1);
9. MonteCarlo (HostCallResult, HostCallConfidence, HostStock, HostStrike, HostTime);
10. #pragma offload_wait target(mic:0) wait(&sigComplete0)
11. #pragma offload_wait target(mic:1) wait(&sigComplete1)

3.5 Heterogeneous Computing with Native Execution on Many-core coprocessors

Simultaneous heterogeneous computing can also be achieved with a throughput model by invoking and running the coprocessor program natively while concurrently running the program on the host processors. The first step is to cross compile the executable file to be executed on the coprocessor. Second, a shell script can create a shell session on the coprocessor and then copy and execute the program. In this model, the scripting language can invoke and overlay the program execution on the host processor and the coprocessors and achieve high throughput performance.

3.6 Performance Summary

Simultaneous heterogeneous computing can achieve high energy efficiency by minimizing the host processor stall state. This is because the host is always consuming power, even when idle and making no contribution to Monte Carlo problem solving, and simultaneous heterogeneous computing eliminates that inefficiency by utilizing the host processors.
Table 3. Performance Gain from Step 5 using 2 Intel® Xeon Phi™ Coprocessor 7120.

<table>
<thead>
<tr>
<th>Step</th>
<th>Before Optimization</th>
<th>After optimization</th>
<th>Gain Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>15.6 opt/sec</td>
<td>359 opt/sec</td>
<td>23.1</td>
</tr>
<tr>
<td>Step 2</td>
<td>359 opt/sec</td>
<td>581 opt/sec</td>
<td>1.62</td>
</tr>
<tr>
<td>Step 3</td>
<td>581 opt/sec</td>
<td>1781 opt/sec</td>
<td>3.07</td>
</tr>
<tr>
<td>Step 4</td>
<td>1781 opt/sec</td>
<td>43.9 kopt/sec</td>
<td>24.9</td>
</tr>
<tr>
<td>Step 5</td>
<td>43.9 kopt/sec</td>
<td>118 kopt/sec</td>
<td>2.69</td>
</tr>
</tbody>
</table>

4 Energy Efficiency and Compute Density

In addition to application performance, we also measured energy efficiency and computational density. Both of these measurements are important for the banking and securities industries.

The Performance to Power ratio was measured for Apollo 6000 servers both with and without two Intel® Xeon Phi™ coprocessors. The program’s energy efficiency is measured as the ratio of Performance/Power for these servers. In both cases the servers had two Intel® Xeon E5-2698 v3 processors.

Interestingly, the power consumption of the server with two Phi coprocessors was almost exactly twice that of the server that had only Xeon processors. Because the server density of these servers is also 2:1, the Performance/Density ratio is the same as the Performance/Power ratio, as seen in the Tables 4 and 5.

4.1 Performance/Power Measurement

Table 4. Performance/Power Gain from Step 5 using the Step-wise Optimization Framework

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes</td>
<td>1.94</td>
<td>1.83</td>
</tr>
<tr>
<td>Binomial</td>
<td>0.99</td>
<td>0.99</td>
</tr>
<tr>
<td>Monte Carlo RNG</td>
<td>1.39</td>
<td>1.32</td>
</tr>
<tr>
<td>MonteCarlo PreGen</td>
<td>3.70</td>
<td>2.30</td>
</tr>
</tbody>
</table>

4.2 Performance/Density Measurement

Table 5. Performance/Density Gain from Step 5 using the Step-wise Optimization Framework

<table>
<thead>
<tr>
<th></th>
<th>SP</th>
<th>DP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Scholes</td>
<td>1.9</td>
<td>1.8</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Monte Carlo RNG</td>
<td>1.4</td>
<td>1.3</td>
</tr>
<tr>
<td>MonteCarlo PreGen</td>
<td>3.7</td>
<td>2.3</td>
</tr>
</tbody>
</table>
5 Summary

In this paper, we took a practical approach to simultaneous heterogeneous computing on a widely available platform, the HP Apollo 6000 System, and we analyzed a few derivative pricing applications popular with investment banks and other financial service firms. We also proposed a structured framework to conceptualize various steps in turning serial and sequential applications into simultaneous heterogeneous applications. The benefit of such a framework is that it allows the developer to realize and quantify substantial performance improvements while concentrating on their application by implementing a straight-forward optimization method. In the case study, we went into detail in how to achieve maximum efficiency in a heterogeneous computing environment using the Monte Carlo method with Intel’s MKL random number generator as an example and Intel Parallel Composer XE 2015 as a development tool. In the end, we demonstrated high performance gains by using widely available software tools and libraries in concert with Intel Xeon Phi coprocessors and applying the Step-wise Optimization Framework. Finally, we also looked at two other limiting factors in the financial service industry, power consumption and rack space. We demonstrated that the performance gain we achieved on the heterogeneous computing platform increases the computation efficiency when these two additional factors are considered.

Chart 1. Heterogeneous Computing using HP Apollo 6000 platform increases computational efficiency in terms of power consumption and rack space.
References