Brown Deer
Technology

Application Note: Programming Parallella Using STDCL

Contents

1 Introduction

The goal of this Application Note is to provide a quick introduction to programming Parallella using STDCL with a few concrete examples that exercise basic features of the programming model provided by the COPRTHR® SDK for Parallella. The STDCL® API is designed for application programmers desiring an intuitive, concise programming model for offloading computations to an accelerators or co-processors. STDCL is designed in a style inspired by conventional UNIX APIs for C programming, and may be used directly in C/C++ applications, with additional support for Fortran through direct API bindings.

The STDCL examples below are portable. The same code that runs on Parallella will work on heterogeneous platforms with AMD CPUs, GPUs and APUs, Nvidia GPUs, Intel CPUs and MIC accelerators, ARM CPUs, and even a Qualcomm Adreno GPU. The only modifications would involve selecting the correct context and device number for a specific platform configuration.

The code below is tutorial in nature and not optimized in any way. Therefore, none of the examples are expected to exhibit good performance. Precisley how to optimize these algorithms is understood. However, the architecture-specific topic of code optimization is not discussed here, and is instead deferred to a future document devoted to the subject. The examples below are the starting point for any optimized implementations.

---

2 Matrix-Vector Multiply: A Simple Example

A simple matrix-vector multiply provides a convenient example for explaining how to offload computations to an accelerator or co-processor. As a first example, there is suficient complexity to exercise the basic steps without complex code that distracts from the concepts and techniques. For most programmers working with Parallella for the first time, the most fundamental question will be how to port code to Parallella that utilizes the Epiphany RISC array co-processor. We begin with a very simple C implementation of a matrix-vector multiply and walk through the porting process.

2.1 Basic C Implementation

The C program below provides a canonical implementation of the algorithm, which we will use as a reference.

```
/* matvecmult_cref.c */

#include <stdio.h>
#include <stdlib.h>

inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i,j;
    unsigned int n = 1024;

    /* allocate memory */
    float* aa = (float*)malloc(n*n*sizeof(float));
    float* b = (float*)malloc(n*sizeof(float));
    float* c = (float*)malloc(n*sizeof(float));

    /* initialize matrix aa[] and vector b[], and zero result vector c[] */
    for(i=0; i<n; i++) for(j=0; j<n; j++) aa[i*n+j] = (1.0/n/n)*i*j*parity(i*j);
    for(i=0; i<n; i++) b[i] = (1.0/n)*i*parity(i);
    for(i=0; i<n; i++) c[i] = 0.0f;

    /* perform calculation */
    for(i=0; i<n; i++) {
        float tmp = 0.0f;
        for(j=0; j<n; j++) tmp += aa[i*n+j] * b[j];
        c[i] = tmp;
    }

    for(i=0;i<n;i++) printf("%d %f %f\n",i,b[i],c[i]);

    free(aa);
    free(b);
    free(c);
}
```

The elements of the program are rather simple. Storage for the matrix and vectors are allocated and initialized. The matrix-vector multiply calculation is performed producing the result vector, which is subsequently printed out.

2.2 Parallella is a Heterogeneous Computing Platform

The Parallella platform is an example of a heterogeneous computing platform and consists of a dual-core ARM CPU and an Epiphany RISC array co-processor. This introduces all of the fun and challenges associated with accelerators or co-processors seen in the evolution of modern HPC architecture. It is important to understand that architecture matters, and Parallella is actually unique. Understanding the unique features, capabilities, and limitations is key to programming the Parallella platform.

Turning back to our matrix-vector multipl example, porting this program to use the Epiphany co-processor on Parallella introduces three basic requirements for modifying the code:

1. Cross-compilation of code to execute on the Epiphany co-processor
2. Distributed memory management[1]
3. Coordination of operations on the Epiphany co-processor

In the following sections these requirements are addressed to produce a program that offloads the matrix-vector multiply calculation to the Epiphany co-processor.

2.3 Kernel Code for the Epiphany Co-Processor

With our simple program we want to move the execution of the parallel portion over to the Epiphany RISC array. This follows a compute offload model typically used with accelerators. The target code is the loop

[1] Parallella supports a unified address space but this is not enabled on the platform by default

body of the for-loop over i identified with the comment "perform calculation". This is precisely the code that which would be targeted with a #pragma parallel for construct using OpenMP. STDCL employs an explicit programming model that provides the programmer with greater control over the interactions between the host and co-processor and the code executed on the co-processor, as compared with implicit models like OpenMP and other #pragma mark-up APIs.

In our example here the code is quite simple but nevertheless we must pull this code fragment out and place it in a separate file so that it may be cross-compiled for Epiphany. We use this for-loop body to create a kernel or thread function as shown below, placing it in the file matvecmult_kern.c.

```
/* matvecmult_kern.cl */

#include <stdcl.h>

void matvecmult_kern( unsigned int n, float* aa, float* b, float* c )
{
    int i,j;

    i = get_global_id(0);

    float tmp = 0.0f;
    for(j=0; j<n; j++) tmp += aa[i*n+j] * b[j];
    c[i] = tmp;
}
```

The construction of kernels or thread functions is not difficult. A key point to understand is that this function will be executed for each thread that is launched in parallel. That is the execution context of the function. In our simple example the parallelism is defined by the range of the for-loop over i. The loop itself is not part of the kernel - this is a key point to understand. Instead one envisions the for-loop over i replaced with the pseudo-code,

```
...
foreach(i)
    matvecmult_kern(n,aa,b,c);
...
```

Upon examining the body of our kernel, the last three lines are exactly those taken from the original for-loop body. The only line that warrants explanation is the second line,

```
i = get_global_id(0);
```

Since the kernel is implicitly executed for each index i - that is the execution context of the kernel, we must have a way for each thread to self-identify with the effective iteration, i.e., value of i, it should be calculating. This is done by getting the global index of the thread in the index space of execution.

2.4 Host Code Modifications

Having created our kernel code, we must now modify the host code to offload the parallel calculation. First we must use memory that is shareable between the ARM host and the Epiphany co-processor. This is accomplished by simply replacing the malloc() calls with clmalloc() calls used to allocate device-shareable memory,

```
float* aa = (float*)clmalloc(stdacc,n*n*sizeof(float),0);
float* b = (float*)clmalloc(stdacc,n*sizeof(float),0);
float* c = (float*)clmalloc(stdacc,n*sizeof(float),0);
```

where `stdacc` is the STDCL context for the Epiphany accelerator, the second argument is just the size of the allocations in bytes, and he flags argument in this case can be left as 0.

Next we must ensure that the memory is synchronized with the Epiphany device prior to its use by our kernel. This type of memory management is inherent to a platform with distributed memory like Parallella. Ensuring memory consistency is a basic requirement for proper operation. Although it is quite possible to create an API premised on automatic memory consistency, such approaches have not been successful thus far with accelerators, and all APIs have inevitably exposed this control to the programmer. STDCL introduces this programming requirement directly and with clean syntax as compred to the implicit syntax found with `#pragma` mark-up APIs.

Modifying our program to ensure the data is synchronized with the device memory requires adding thre `clmsync()` calls,

```
clmsync(stdacc,0,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
clmsync(stdacc,0,b,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
clmsync(stdacc,0,c,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
```

The flag `CL_EVENT_NOWAIT` causes the clmsync() call to be non-blocking and return immediately. As a consequence we will later need to use a barrier call on the host to wait until all asynchronous operations have been completed.

Now we can offload the computation. This is done in two steps. First we specify an index range for the parallel computation based on the range of the original for-loop over `i`,

```
clndrange_t ndr = clndrange_init1d( 0, n, 16 );
```

This specifies an index space of `n` to be executed using a local workgroup of 16 that maps to the 16 RISC cores on the Epiphany device. We then fork the execution of the kernel on the Epiphany device. This is a fork since the host code will continue asynchronously as the parallel threads of execution are launched on the co-processor,

```
clexec(stdacc,0,&ndr,matvecmult_kern,n,aa,b,c);
```

Notice here that the kernel symbol, `matvecmult_kern`, is the name of our kernel function. Additionally, the arguments passed to the kernel are `n,aa,b,c` specified as the last arguments to the `clexec()` call. The ‘clexec() call is non-blockinh and therefore it is required that the host code eventually block until all device operations are completed.

Next we must ensure that the device memory where the results are stored is synchronized with the host memory using the same `clmsync()` call used above, but with a different flag,

```
clmsync(stdacc,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);
```

The last thing we need to do is block on the host until all of the asynchronous operations on the device are compelted. This is done with the `clwait()` call,

```
clwait(stdacc,0,CL_ALL_EVENT);
```

Following this host-device synchronization call the results will be guaranteed to be accessible on the host and may be printed out as in the original program.

Putting this all together, the modified host program for the host is shown below.

```
/* matvecmult_host.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdcl.h>

inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i,j;
    unsigned int n = 1024;

    /* allocate device-shareable memory */
    float* aa = (float*)clmalloc(stdacc,n*n*sizeof(float),0);
    float* b = (float*)clmalloc(stdacc,n*sizeof(float),0);
    float* c = (float*)clmalloc(stdacc,n*sizeof(float),0);

    /* initialize matrix aa[] and vector b[], and zero result vector c[] */
    for(i=0; i<n; i++) for(j=0; j<n; j++) aa[i*n+j] = (1.0/n/n)*i*j*parity(i*j);
    for(i=0; i<n; i++) b[i] = (1.0/n)*i*parity(i);
    for(i=0; i<n; i++) c[i] = 0.0f;

    /* sync data with device memory */
    clmsync(stdacc,0,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    clmsync(stdacc,0,b,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    clmsync(stdacc,0,c,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

    /* perform calculation */
    clndrange_t ndr = clndrange_init1d( 0, n, 16 );
    clexec(stdacc,0,&ndr,matvecmult_kern,n,aa,b,c);

    /* sync data with host memory */
    clmsync(stdacc,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

    /* block until co-processor is done */
    clwait(stdacc,0,CL_ALL_EVENT);

    for(i=0;i<n;i++) printf("%d %f %f\n",i,b[i],c[i]);

    clfree(aa);
    clfree(b);
    clfree(c);
}
```

2.5 Program Compilation

Using the COPRTHR `clcc` compiler tools, building the program is easy and follows a standard compilation model. The kernel code is first compiled using `clcc` and then the host program can be compiled with the kernel code being directly linked to create a single executable program,

```
] clcc -k -c matvecmult_kern.cl
] gcc -o matvecmult.x matvecmult_host.c matvecmult_kern.o -I$COPRTHR_PATH/include \
    -L$COPRTHR_PATH/lib -lstdcl -locl
```

Here we assume that the environment variable `$COPRTHR_PATH` is set to the path where the COPRTHR SDK was installed. By default this location will be `/usr/local/browndeer`.

We can verify that the kernel code is linked in using the `clnm` command,

```
] clnm matvecmult.x
clnm: 'matvecmult_kern.c' bin [coprthr:ARMv7]
clnm: 'matvecmult_kern.c' bin [coprthr:E16G Needham]
clnm: 'matvecmult_kern.c' ksym matvecmult_kern
clnm: 'matvecmult_kern.c' src [<generic>]
```

We can then run the program that offloads the parallel computation to the Epiphany co-processor,

```
] ./matvecmult.x
```

2.6 Debug support

What to do if things go wrong? There is a debug interface that can be very useful. If the program is not running correctly, it can be useful to run the program enabling clmesg debug output,

```
] cldebug -- ./matvecmult.x
```

Expect that the output will be very verbose and lengthy, so redirecting to a temporary log file is helpful.

If an error occurs when compiling the kernel code using clcc, the same debug interface can be used to try to track down the problem,

```
] cldebug -- clcc -k -c matvecmult_kern.c
```

3 Matrix-Vector Multiply Revisited: Architecture Matters

The previous example showed how to port a simple matrix-vector-multiply code to use the Epiphany co-processor on Parallella. The implimentation is functinally correct, but glosses over one of the first concerns any programmer should have when programing Parallella. The question is not how one can organize the parallelism, but rather how one should organize the parallelism, given the realities of the underlying architecture.

At issue is the fact that our first pass at porting to Epiphany contains a convention suitable for, and inherited from, algorithms targeting GPUs. A GPU benefits from, and requires for efficiency, the use of heavily multi-threads algorithms with minimum parallelism on the order of thousands of threads. The Epiphany RISC array is a scalar multi-core processor with no multi-threads support. The idea of launching thousands of parallel threads on an 16-core Epiphany processor is not only naive, it risks incurring undue overhead costs. As pointed out in the previous section, there should be no surprises over the performance of our first attempt at porting to Epiphany. Nor will the following modification bring much in the way of a measureable improvement. However, it is an important modification for programmers learning to program Parallella because it shows the correct manner in which to organize the parallelism.

The basic problem with our initial port is exemplified by two related observations. First, when we defined the index range used to launch threads, we asked for n threads executed in local workgroups of 16to match the number of cores on the Epiphany processor. Second, when we examime our kernel code, we find that we have a very "light" kernel that perofrms a single summation over n products.

Given the overhead of launching executable code on the Epiphany co-processor, one should design the kernels to perform as much work as possible. This design goal is also true when programming GPUs, but it is trumped by the requirement for thousands of threads. Epiphany is not a GPU and this alters the fundamental approach that must be taken.

Based on our architecture, we know how we would like the program to behave. Specifically, we would like to perform our parallel calculation using 16 threads corresponding to the 16 phisial RISC array cores. Any over-threading beyond this cannot bring any benefit and can only incur penalties for increased overhead. So we take this into account as a basic design strategy when porting our program to Parallella.

The code modifications we need to make involve "folding" partial loops over i into our kernel so that each of the 16 physical cores performs (roughly) 1/16th of the computational work within a single kernel so that only 16 threads need to be launched and is conceptually more "correct" for performing the calculation on the Epiphany RISC array.

3.1 Modifying the Kernel Code

The kernel from our first attempt is modified by add a loop over i inside the kernel itself with a range defined such that each kernel performs (roughly) 1/16th of the computational work in calculating c[i]. The workload is not guaranteed to be exactly evenly distributed unless the total vector size n is commensurate with 16. However the distribution is as close to even for the non-comensurate cast as possible. The complete modifications are shown below for our second matrix-vector-multiply kernel.

```
/* matvecmult_kern2.cl */

#include <stdcl.h>

void matvecmult_kern2( unsigned int n, float* aa, float* b, float* c )
{
    int i,j,k;

    k = get_global_id(0);
    int n16 = n/16;
    int m16 = n%16;

    int ifirst = k*n16 + ((k>m16)? 0:k);
    int iend = ifirst + n16 + ((k<m16)? 1:0);

    for(i=ifirst; i<iend; i++) {
        float tmp = 0.0f;
        for(j=0; j<n; j++) tmp += aa[i*n+j] * b[j];
        c[i] = tmp;
    }
}
```

3.2 Modifying the Host Code

The modifications to the host code are much simpler, and in fact our trivial. The only required change is to modify the index range over which the kernel is executed, changing this from n to 16,

```
clndrange_t ndr = clndrange_init1d( 0, 16, 16 );
```

For completeness, the full host code for our second version of the matrix-vector multiply code is shown below.

```
/* matvecmult_host2.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdcl.h>

inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i,j;
    unsigned int n = 1024;

    /* allocate device-shareable memory */
    float* aa = (float*)clmalloc(stdacc,n*n*sizeof(float),0);
    float* b = (float*)clmalloc(stdacc,n*sizeof(float),0);
    float* c = (float*)clmalloc(stdacc,n*sizeof(float),0);

    /* initialize matrix aa[] and vector b[], and zero result vector c[] */
    for(i=0; i<n; i++) for(j=0; j<n; j++) aa[i*n+j] = (1.0/n/n)*i*j*parity(i*j);
    for(i=0; i<n; i++) b[i] = (1.0/n)*i*parity(i);
    for(i=0; i<n; i++) c[i] = 0.0f;

    /* sync data with device memory */
    clmsync(stdacc,0,aa,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    clmsync(stdacc,0,b,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    clmsync(stdacc,0,c,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
    /* perform calculation */
    clndrange_t ndr = clndrange_init1d( 0, 16, 16 );
    clexec(stdacc,0,&ndr,matvecmult_kern2,n,aa,b,c);

    /* sync data with host memory */
    clmsync(stdacc,0,c,CL_MEM_HOST|CL_EVENT_NOWAIT);

    /* block until co-processor is done */
    clwait(stdacc,0,CL_ALL_EVENT);

    for(i=0;i<n;i++) printf("%d %f %f\n",i,b[i],c[i]);

    clfree(aa);
    clfree(b);
    clfree(c);
}
```

3.3 Compile and Run

The modified program can then be compiled and executed using the following commands,

```
] clcc -k -c matvecmult_kern2.cl
] gcc -o matvecmult2.x matvecmult_host2.c matvecmult_kern2.o -I$COPRTHR_PATH/include \
    -L$COPRTHR_PATH/lib -lstdcl -locl
] ./matvecmult2.x
```

4 Reduction Example

As a second example we consider a reduction algorithm which may serve as a proxy for an entire class of algorithms in which a single scalar result must be calculated over an array of values. Here we will simply add the elements of a vector.

4.1 Simple C Implementation

The C program below implements a simple reduction in which the elements of a single vector are added to produce a scalar sum. This code will be used as the starting point for offloading the reduction to the Epiphany co-processor.

The program is relatively trivial and consists of allocating memory for the vector, initializing the vector, summing the elements of the vector and printing the result which is a single value.

```
/* reduction_cref.c */

#include <stdio.h>
#include <stdlib.h>
```

```
inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i;

    unsigned int n = 1000001;

    /* allocate data vector */
    float* data = (float*)malloc(n*sizeof(float));

    /* initialize data vector */
    for(i=0; i<n; i++) data[i] = 2.2f*i*parity(i);

    /* perform calculation */
    float sum = 0;
    for(i=0; i<n; i++) {
        sum += data[i];
    }

    printf("sum %f\n",sum);
}
```

4.2 Kernel Code

The parallelism in the reference code is found in the for-loop over `i` used to perofrm the calculation of the sum over elements. The loop body, here consisting of only a single line of code, must be pulled out and used to create a kernel for parallel execution on the Epiphany device.

The approach discussed in Section 3 in which the parallelism is matched the the phyical cores of the RISC array will be employed here from the start. The idea is quite simple. The work of adding the elements of the data vector will be ditributed as evenly as possible over the 16 phyisical cores to calculate partial sums. Subsequently the 16 partial sums will be added together to calculate the final result.

The kernel that accomplishes this is shown below. The bookkeeping used to determine the range of the sum defined by `ifirst` and `iend` for a given core `k` will be left as an exercise (one the reader is encouraged to work through).

The arguments to the kernel are the total number of elements `n` and the data vector `data`. The third argument is a temporary array used to store the partial sums and will therefore have a length of 16 when allocated on the host.

```
/* reduction_kern.cl */

#include <stdcl.h>

void reduction_kern( unsigned int n, float* data, float* temp )
{
    int i,k;

    k = get_global_id(0);
    int n16 = n/16;
    int m16 = n%16;

    int ifirst = k*n16 + ((k>m16)? m16:k);
    int iend = ifirst + n16 + ((k<m16)? 1:0);

    float sum = 0;
    for(i=ifirst; i<iend; i++) {
        sum += data[i];
    }

    temp[k] = sum;
}
```

4.3 Host code

Beginning with the reference code, modifications analogous to those described in Section 2 are introduced. Briefly, we must change the memory allocation so as to allocate device-shareable memory, synchronize the memory between host and device at the appropriate steps, fork the execution of the kernel above, and finally block until all co-processor device operations are complete.

Additionally, a small calculation is added at the end of the program to add the partial sums to produce a final result. It is reasonable, though somewhat inellegant, to perofrm this final sum on the host. If we have a large enough data vector to warrant the use of a parallel RISC array, performing 16 addition operations on the host wold not be expected to cause any noticeable performance penalty. In fact, thi smay be the more efficient approach.

```
/* reduction_host.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdcl.h>

inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i;
    unsigned int n = 1000001;

    /* allocate data vector */
    float* data = (float*)clmalloc(stdacc,n*sizeof(float),0);

    /* initialize data vector */
    for(i=0; i<n; i++) data[i] = 2.2f*i*parity(i);

    /* sync data to device memory */
    clmsync(stdacc,0,data,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

    /* allocate array to store partial sum */
    float* temp = (float*)clmalloc(stdacc,16*sizeof(float),0);
    /* perform calculation */
    clndrange_t ndr = clndrange_init1d( 0, 16, 16);
    clexec(stdacc,0,&ndr,reduction_kern,n,data,temp);

    /* sync partial sums to host memory */
    clmsync(stdacc,0,temp,CL_MEM_HOST|CL_EVENT_NOWAIT);

    /* block until all co-processor operations are complete */
    clwait(stdacc,0,0);

    /* perform final reduction over partial sums on the host */
    float sum = 0;
    for(i=0; i<16; i++) sum += temp[i];

    printf("sum %f\n",sum);
}
```

4.4 Compile and Run

Thanks to the COPRTHR clcc tools, compiling the program is simple and may be accomplished with the following commands,

```
] clcc -k -c reduction_kern.cl
] gcc -o reduction.x reduction_host.c reduction_kern.o -I$COPRTHR_PATH/include \
    -L$COPRTHR_PATH/lib -lstdcl -locl
] ./reduction.x
```

4.5 Using Barriers to Synchronize Threads

In the reduction code above we perform the final summation over partial sums on the host because it was convenient and negligible in terms of performance to do so. However, it would be nice for completeness to be able to perform the entire reduction on the co-processor. This can be accomplished by modifying the kernel so that the final summation over 16 partial sums is carried out on thread 0. Since this final summation should only be done after all of the other threads have completed their partial sums, a barrier is needed to synchronize the threads. The complete modification to the kernel code is shown below.

```
/* reduction_kern2.cl */

#include <stdcl.h>

void reduction_kern2( unsigned int n, float* data, float* temp )
{
    int i,k;

    k = get_global_id(0);

    int n16 = n/16;
    int m16 = n%16;
    int ifirst = k*n16 + ((k>m16)? m16:k);
    int iend = ifirst + n16 + ((k<m16)? 1:0);

    float sum = 0;
    for(i=ifirst; i<iend; i++) sum += data[i];

    if (k>0)  {
        temp[k] = sum;
        barrier(0);
    } else {
        barrier(0);
        for(i=1; i<16; i++)
            sum += temp[i];
        temp[0] = sum;
    }
}
```

The only modification required for the host code is the elimination of the loop that performed this final sum on the host, and replacing it with code that identifies the result as the value saved in temp[0]. The modified host code is shown below.

```
/* reduction_host2.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdcl.h>

inline int parity( int x ) { return ( (x%2)? +1 : -1 ); }

int main()
{
    int i;

    unsigned int n = 1000001;

    /* allocate data vector */
    float* data = (float*)clmalloc(stdacc,n*sizeof(float),0);

    /* initialize data vector */
    for(i=0; i<n; i++) data[i] = 2.2f*i*parity(i);

    /* sync data to device memory */
    clmsync(stdacc,0,data,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

    /* allocate array to store partial sum */
    float* temp = (float*)clmalloc(stdacc,16*sizeof(float),0);

    /* perform calculation */
    clndrange_t ndr = clndrange_init1d( 0, 16, 16);
    clexec(stdacc,0,&ndr,reduction_kern2,n,data,temp);

    /* sync partial sums to host memory */
    clmsync(stdacc,0,temp,CL_MEM_HOST|CL_EVENT_NOWAIT);
    /* block until all co-processor operations are complete */
    clwait(stdacc,0,0);

    float sum = temp[0];

    printf("sum %f\n",sum);
}
```

The modified reduction code may be compiled with the following commands,

```
] clcc -k -c reduction_kern2.cl
] gcc -o reduction2.x reduction_host2.c reduction_kern2.o -I$COPRTHR_PATH/include \
    -L$COPRTHR_PATH/lib -lstdcl -locl
] ./reduction2.x
```

---

5 Stencil 2D Example

As a final example, we consider a 9-point stencil algorithm for a Gaussian filter. The elements of the calculation are shown in the C reference code below. A 2D array of size 1000x1000 is filled with synthetic values and then a 9-point stencil filter is applied repeatedly `ncount` times using a double-buffer rotation scheme in which the data_in and data_out pointers are swapped each iteration. A final checksum is calculated as a proxy of the result.

5.1 Basic C Implementation

```
/* stencil2d_cref.c */

#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;
    int x,y;

   int width = 1000; /* multiple of 4 */
   int height = 1000; /* multiple of 4 */
   int ncount = 10;

    /* define stencil coefficients */
   float stendiag = 0.00125;
   float stennext = 0.00125;
   float stenctr = 0.99;

    /* allocate 2D arrays */
    float* data_in = (float*)malloc(sizeof(float)*width*height);
   float* data_out = (float*)malloc(sizeof(float)*width*height);

    /* initialize arrays */
   for(y=0; y<height; y++)
    for(x=0; x<width; x++) {
         data_in[x+y*width] = 1.234f * ((x+y)%211);
         data_out[x+y*width] = 0;
   }

   for(i=0; i<ncount; i++) {

        /* apply stencil */

    for(y=1; y<height-1; y++)
    for(x=1; x<width-1; x++) {

        int c = x+y*width;

        data_out[c] = stendiag * data_in[c-width-1]
            + stennext * data_in[c-width]
            + stendiag * data_in[c-width+1]
            + stennext * data_in[c-1]
            + stenctr * data_in[c]
            + stennext * data_in[c+1]
            + stendiag * data_in[c+width-1]
            + stennext * data_in[c+width]
            + stendiag * data_in[c+width+1];
    }

        /* swap data buffers */
    float* ptmp = data_in;
    data_in = data_out;
        data_out = ptmp;
   }

   float sum;
   for(y = 0; y<height; y++)
    for(x = 0; x<width; x++) {
      sum += data_out[x+y*width];
    }
   printf("checksum %f\n",sum);

    free(data_in);
   free(data_out);
}
```

5.2 Kernel Code

The parallelism in the reference code is found in the double for-loop over x and y in which the stencil is applied to each element in the 2D array. Following the approach used with previous kernels, we will construct a kernel designed to be executed by each of the 16 RISC cores. One difference will be in the topology of the mapping of those cores. Rather than map the problem to a 1D array of 16 RISC cores, we will map the problem to a 2D array of 4x4 RISC cores, since this matches the dimensionality of the 2D stencil algorithm.

For simplicity we will assume that the width and height of our data arrays are a multiple of 4. Treating more general sizes is not too complicated, but introduces sufficient extra bookkeeping to become distracting, so we leave this as an exercise.

The kernel below will apply the stencil operation to a tile (approximately) 1/16th the size of the data array. The bookkeeping implements the constraint that elements on the outer edge of the array is not calculated since the stencil leaves the update of these points ill-defined.

```
/* stencil2d_kern.cl */

#include <stdcl.h>

/* note: width and height are assumed to be a multiple of 4 for simplicity */

void stencil2d_kern(
    float* data_in, float* data_out,
    const int width, const int height,
    const float stenctr, const float stennext, const float stendiag
)
{
    int i,j;

   i = get_global_id(0);
   j = get_global_id(1);

    int w4 = width/4;
    int xfirst = i*w4;
    int xend = xfirst + w4;
    if (xfirst==0) xfirst = 1;
    if (xend==width) xend -= 1;

    int h4 = height/4;
    int yfirst = j*h4;
    int yend = yfirst + h4;
    if (yfirst==0) yfirst = 1;
    if (yend==height) yend -= 1;

    int x,y;

   for(y=yfirst; y<yend; y++)
   for(x=xfirst; x<xend; x++) {

    int c = x+y*width;

      data_out[c] = stendiag * data_in[c-width-1]
         + stennext * data_in[c-width]
         + stendiag * data_in[c-width+1]
         + stennext * data_in[c-1]
         + stenctr * data_in[c]
         + stennext * data_in[c+1]
         + stendiag * data_in[c+width-1]
         + stennext * data_in[c+width]
         + stendiag * data_in[c+width+1];
   }

}
```

5.3 Host Code

The host code can be derived from the reference code above following the same general steps used in the previous examples. Memory allocation is modified for allocating device-shareable memory using `clmalloc`. Memory consistency is managed using `clmsync()` calls. The kernel will be executed using 16 threads corresponding to the 16 RISC cores on the co-processor device using `clexec()`, and `clwait()` is used to block on the host until all device operations on the co-processor are complete. All of this is implemented in the host code below.

```
/* stencil2d_host.c */

#include <stdio.h>
#include <stdlib.h>
#include <stdcl.h>

int main()
{
   int i;
    int x,y;

   int width = 1000; /* multiple of 4 */
   int height = 1000; /* multiple of 4 */
   int ncount = 10;

    /* define stencil coefficients */
   float stendiag = 0.00125;
   float stennext = 0.00125;
   float stenctr = 0.99;

    /* allocate 2D arrays */
    float* data_in = (float*)clmalloc(stdacc,sizeof(float)*width*height,0);
   float* data_out = (float*)clmalloc(stdacc,sizeof(float)*width*height,0);

    /* initialize arrays */
   for(y=0; y<height; y++)
    for(x=0; x<width; x++) {
         data_in[x+y*width] = 1.234f * ((x+y)%211);
         data_out[x+y*width] = 0;
   }

    /* sync data with device memory */
   clmsync(stdacc,0,data_in,CL_MEM_DEVICE|CL_EVENT_NOWAIT);
   clmsync(stdacc,0,data_out,CL_MEM_DEVICE|CL_EVENT_NOWAIT);

   for(i=0; i<ncount; i++) {
        /* apply stencil */
    clndrange_t ndr = clndrange_init2d( 0,4,4, 0,4,4 );
    clexec(stdacc,0,&ndr,stencil2d_kern,
        data_in,data_out,width,height,stenctr,stennext,stendiag);

        /* swap data buffers */
    float* ptmp = data_in;
    data_in = data_out;
        data_out = ptmp;
   }

    /* sync data with host memory */
   clmsync(stdacc,0,data_out,CL_MEM_HOST|CL_EVENT_NOWAIT);

    /* block until co-processor is done */
   clwait(stdacc,0,CL_ALL_EVENT);

   float sum;
   for(y = 0; y<height; y++)
    for(x = 0; x<width; x++) {
      sum += data_out[x+y*width];
    }
   printf("checksum %f\n",sum);

    clfree(data_in);
   clfree(data_out);
}
```

5.4 Compile and Run

The stencil2d code may be compiled with the following commands,

```
] clcc -k -c stencil2d_kern.cl
] gcc -o stencil2d.x stencil2d_host.c stencil2d_kern.o -I$COPRTHR_PATH/include \
    -L$COPRTHR_PATH/lib -lstdcl -locl
] ./stencil2d.x
```

---

6 More Information

More information on the COPRTHR SDK and the STDCL API may be found in the COPRTHR Primer and the STDCL API Reference.

Document revision 1.6.0.0