ICHEC White Paper
July, 16th, 2015

An OpenCL design of the Bob Jenkins *lookup3* hash function using the Xilinx SDAccel$^{\text{TM}}$Development Environment

Christian Lalanne, Servesh Muralidharan and Michael Lysaght

Abstract
We report on the implementation of an OpenCL-based design of the Bob Jenkins *lookup3* hash function application on the Virtex 7-based ADM-PCIE-7V3 platform using the new Xilinx SDAccel$^{\text{TM}}$development environment. In particular, we demonstrate how to exploit features of the OpenCL programming model available through SDAccel to improve performance by up to 3.5x relative to a naive OpenCL design of the application. Our early-stage results strongly indicate that near-optimal high performance, low-energy software defined solutions on FPGAs can be delivered using SDAccel's OpenCL development environment with much shorter turnaround times than through hardware-centric RTL flows.

Keywords
OpenCL, FPGA

Irish Centre for High End Computing

Contents

List of Figures

List of Listings

List of Tables

Introduction

In this report, we present an OpenCL-based design of a hashing function which forms a core component of memcached [1], a distributed in-memory key-value store caching layer widely used to reduce access load between web servers and databases.

Our work has been inspired by recent research investigations on dataflow architectures for key-value stores that can sustain a consistent 10Gbps line-rate and which bring significant latency benefits through tight coupling of network interface, memory and compute resources [2]. At the heart of a key-value store architecture, such as memcached, is a hash table, which in essence determines the memory address of a value as a function of an incoming key. This is achieved by first applying the chosen hash function to the contents of a key to produce an address in the table. From this location, a pointer to the address within the value storage area can be retrieved.

In the work reported here, we focus on the hashing function stage only. In particular, we have focused on the OpenCL design of the Bob Jenkins *lookup3* hash function [3], which is used in the open source software version of memcached and is well known to work effectively over a broad range of key types. This function processes variable sized keys iteratively in 96bit chunks and each chunk is split into three 32bit numbers, which are added to a set of state variables. Before the next chunk is read, these state variables are mixed using addition, subtraction and XOR operations. Due to the inherent feedback loop, the hash function cannot be easily pipelined.

On a broader basis, our investigations have been stimulated by the recent release of the Xilinx SDAccel development environment [4], which supports OpenCL-based design of energy efficient high throughput solutions on FPGAs. Indeed, much of the focus of this report is on demonstrating how we have been able to achieve significant performance gains by exploiting the high-level parallel abstractions afforded through SDAccel's OpenCL compiler.

The rest of this report is organized as follows: In section 1 we provide a short overview of the sequential Bob Jenkins hashing function, which acts as the starting point for our discussion around OpenCL-based optimisations. Section 2 provides an overview of our first "naive" OpenCL hashing function kernel which we evaluate for the first time on the Xilinx Virtex 7-based ADM-PCIE-7V3. In section 3 we provide an overview of how we have optimised the OpenCL hashing function kernel further in order to achieve a performance improvement of ∼3.5x relative to the "naive" OpenCL kernel and briefly discuss how we have further implemented lower-level optimisations to achieve a performance improvement of up ∼187x relative to the naive OpenCL hashing kernel. Finally, we discuss some key conclusions and plans for future work.

1. The sequential Jenkins lookup3 hash function

In this section we provide a brief overview of the sequential Bob Jenkins *lookup3* hashing function [3], which is freely available in the public domain. This hash function, supports a maximum key size of 256 bytes, but in our implementation we use keys up to a size of 60 bytes. A sequential implementation of the Bob Jenkins *lookup3* hash function, `hashlittle`, written in C is presented in listing 1. The algorithm has three fundamental stages:

- Combining key length and initialisation value to set up an initial state.
- The mixing of the bits of the keys in 12 byte increments.
- The processing of the remaining bytes of the key.

Listing 1. The Bob Jenkins `lookup3` function implemented with C.

```

uint32_t hashlittle( const void *key, size_t length, uint32_t initval){

    /* setting init state */
    uint32_t a,b,c;
    a = b = c = 0xdeadbeef + ((uint32_t)length) + initval;

    const uint32_t *k = (const uint32_t *)key;

    /* mixing */
    while (length > 12)
    {
      a += k[0];
      b += k[1];
      c += k[2];

      mix(a,b,c);

      length -= 12;
      k += 3;
    }

    /* mix remainder */
    return mixRemainder(a, b, c, k, length);
}
```

The `hashlittle` function in listing 1 computes the hash of a single key for a given length and initialisation value. The function in effect reduces the key length by 12 bytes for each mixing iteration. The most computationally expensive part of the function is the mixing of the bits of a given key. Once a given key reaches a length of less or equal than 12 bytes the remaining bits are extracted and mixed within the `mixRemainder()` function.

2. A first OpenCL design of the Jenkins *lookup3* hash function

In this section, we describe our first design strategy to enable the *lookup3* function on the FPGA using SDAccel (all results in this report have been obtained using SDAccel v2014.3.5). Taking the C-based function in listing 1 as our base implementation, our first step in designing an OpenCL implementation of the *lookup3* function was to identify the full list of input arguments and the return values for the OpenCL kernel function. For our initial parallel design, we began with the strategy that each OpenCL `Work Item` launched will process one key, and therefore, the number of `Work Items` launched should logically equal the number of keys that need to be hashed. The arguments for our initial OpenCL kernel function are therefore:

- `keys`: Each key is stored within contiguous blocks of memory. The size of these blocks is equal to the maximum key size. A buffer consisting of these key blocks forms the input key array to be processed.
- `lengths`: The buffer consisting of lengths of the keys.
- `initvals`: The buffer consisting of initialization values of the keys.
- `out`: The buffer to store the hash value computed by the kernel.

Listing 2. A naive OpenCL kernel function

```

__kernel void hashlittle(__global const uint* restrict keys,
                         __global const uint* restrict lengths,
                         __global const uint* restrict initvals,
                         __global uint* restrict out){

    const uint id = get_global_id(0);

    const uint length = lengths[id];
    const uint initval = initvals[id];
    __global uint* k = &keys[id*15];

    /* setting initial state */
    uint a,b,c;
    a = b = c = 0xdeadbeef + ((uint)length) + initval;

    /* mixing */
    while (length > 12){
        a += k[0];
        b += k[1];
        c += k[2];

        mix(a,b,c);

        length -= 12;

        k += 3;
    }

    /* mix remainder */
    out[id] =  mixRemainder(a, b, c, k, length);
}
```

Each OpenCL `Work Item` is effectively assigned a `key/length/initvals` and a location to store the hash value (`out`) according to the "global id" of the `Work Item`. The majority of our design considerations at this stage were focused on specifying the read/write access patterns from global (off-chip memory) for each `Work Item`. Since SDAccel supports this feature, it allowed us to implement and evaluate an initial OpenCL version of *lookup3* on the FPGA platform with relative ease. The initial OpenCL kernel function is shown in listing 2, where henceforth this OpenCL kernel is referred to as **Kernel1**.

Table 1 shows the time (in milliseconds) to execute this naive OpenCL kernel for 1, 2, 4 and 8 million keys. The **Kernel1** OpenCL implementation was evaluated using a range of `Work Group` sizes, where each `Work Group` consists of a given number of `Work Items`, specified in the table and throughout the report by *WGsize*. The observed increase in performance seen in Table 1 is due to the reduction in the number of `Work Groups` required to process the workload , as the size of each `Work Group` increases (i.e., the increased parallelism exposed).

	Time [ms]		
Kernel1 [2]	**1M**	**2M**	**4M**
WGsize = 1	34513.66	69171.53	134968.23
WGsize = 4096	2092.25	4178.39	8384.41

Table 1. FPGA Naive kernel execution times

It can be seen that, for the case of 1M keys, with our naive OpenCL kernel, we achieve a speedup of ∼16x going from a *WGsize* of 1 to a *WGSize* of 4096.

3. Further OpenCL optimisations of the lookup3 function

In this section we provide an overview of how we have exploited OpenCL to carry out further optimisations of the *lookup3* function running on the FPGA. By far, the most significant improvements in performance are a result of effective use of OpenCL *vector types* in combination with a judicious use of if/else branching, which, in effect, reduces the number of memory transactions to off-chip memory on the FPGA platform. As well as these optimisations, we have included additional optimisations recommended by the SDAccel User Guide [5], which we mention here first as these are included as part of further vector-type-based optimisations that we discuss in more detail below.

3.0.1 Static OpenCL `Work Group` dimension at compile time

Fixing the OpenCL `Work Group` dimensions at compile time is a recommended optimization in the SDAccel User Guide [5], where, by determining loop trip counts, the circuit can be optimized for a given `Work Group` dimension. Listing 3 demonstrates how to apply this optimization in SDAccel using the attribute `reqd_work_group_size` specified by the OpenCL standard [6].

Listing 3. Specifying compile time work group size (WGsize)

```
__kernel void
__attribute__((reqd_work_group_size(4096, 1, 1)))
hashlittle(...){ }
```

3.0.2 Pipelining `Work Items`

In Listing 4 and implementations thereafter, we wrap the body of the kernel in the SDAccel attribute `__attribute__((xcl_pipeline_`
which indicates to SDAccel to pipeline `Work Items` instructions inside of each and every `Work Group` [5].

3.0.3 Maximum number of memory ports

By design, SDAccel will aim to make the most efficient use of resources on the device, often minimising overall resource utilisation in the process. By default, SDAccel will therefore currently use one memory port to access off-chip memory for all the arguments of the kernel, which can result in a performance bottleneck when accessing off-chip memory. By explicitly specifying the maximum number of memory ports available with `set_property max_memory_ports true [get_kernels <kernel name>]` at compile time, SDAccel subsequently instantiates one memory port per kernel argument that is decorated as a global buffer(`__global`), which in some cases, can increase performance. [5].

3.0.4 Vector types

In listing 2 we have employed the `uint` datatype for all the arguments of the kernel, which results in fetches of 4 bytes or 32 bits per memory transaction to off-chip memory, resulting in sub-optimal performance. In order to improve performance, we have modified the **Kernel1** to support OpenCL vector types for the first time, where the maximum vector width supported by SDAccel is 512 bits or 64 bytes. The vector implementation of our kernel uses a key with a maximum of 60 bytes in size, making it possible to fit an entire key within 64 bytes (with padding of the last 4 bytes). As a result, an entire key can be transferred from off-chip global memory in a single memory transaction. However, OpenCL vector types cannot be used in the kernel as found in listing 2 directly as the elements in the vector data type have to be addressed explicitly. To achieve this we unroll the loop manually by switching to an ‘`if`’ branch for every iteration of the loop in listing 2, where the resulting kernel is shown in listing 4. For each access we load an entire key as a vector of sixteen elements and compute parts of the key in each of the ‘`if`’ branches. This implementation will hereafter be referred as **Kernel2**.

Listing 4. OpenCL-optimised hashing kernel function (Kernel2)

```
__kernel void __attribute__((reqd_work_group_size(4096, 1, 1)))
hashlittle(__global const uint16* restrict keys, ... , __global uint16* restrict out){
    __attribute__((xcl_pipeline_workitems)){
        const uint index = get_global_id(0);

        size_t length = read16(lengths[index/16], id%16);
        uint initval = read16(initvals[index/16], id%16);

        setupInitialState(a, b, c);

        if(length>12){
            a += keys[index].s0;
            b += keys[index].s1;
            c += keys[index].s2;

            mix(a,b,c);

            length -= 12;
            ++rem;
        }

        ....

        if(length>12){
            a += keys[index].s9;
            b += keys[index].sa;
            c += keys[index].sb;

            mix(a,b,c);

            length -= 12;
            ++rem;
        }

        c = mixVectorReminder(a, b, c, index, length);
            write16(out, index/16, index%16, c);
    }
}
```

We find that while vector types allowed us to read more data for each transaction an analysis of latency reports suggests that effective pipelining of the loop is impeded. To test this further we switched to employing ‘if/else’ conditional branching that explicitly dictates where the code will exit, allowing for the feeding of data into the pipeline in every cycle, irrespective of the key size. The resulting kernel is shown in listing 5 and will be referred to as **Kernel3**.

Listing 5. OpenCL-optimised hashing kernel function (Kernel3)

```
__kernel void __attribute__((reqd_work_group_size(4096, 1, 1)))
hashlittle(__global const uint16* restrict keys, ... , __global uint16* restrict out){
    const uint index = get_global_id(0);

    __attribute__((xcl_pipeline_workitems)){

        /*reading from specific positions in vector types*/
        setupInitialState(a, b, c);

        bool done = false;
        if(length > 12){
                a += keys[index].s0;
                b += keys[index].s1;
                c += keys[index].s2;

                mix(a,b,c);
                length-=12;
        } else {
            done = true;
            rem0 = keys[index].s0;
            rem1 = keys[index].s1;
            rem2 = keys[index].s2;
        }

        ...

        if(length > 12) {
                a += keys[index].sc;
                b += keys[index].sd;
                c += keys[index].se;

                mix(a,b,c);
                length-=12;
        } else {
            if(!done) {
                done = true;
                rem0 = keys[index].sc;
                rem1 = keys[index].sd;
                rem2 = keys[index].se;
            }
        }

        uint tmp = mixVectorReminder(a,b,c,rem0,rem1, rem2, length);
            write16(out, index/16, index%16, tmp);
    }
}
```

As mentioned, the vector implementation of the kernel uses a key of maximum 60 bytes in size, which makes it possible to fit an entire key within 64 bytes. In this way an entire key can be transferred from off-chip global memory in a single memory transaction. However, when combining different bitwidth reads of variables such as length (4 bytes) and the keys (64 bytes), asymmetric memory transactions can prevent new data being fed in every cycle. While it is straightforward to apply 64 byte memory reads to the keys, this is not the case for lengths or initialisation values. In order to employ the same kernel with minimal modification to use vector types, replication of the length and initialisation values to occupy an entire vector datatype was required. Therefore, length and initialization values were also transferred as 64 byte memory transactions, where only the first 4 bytes of each 64 byte vector were used. Listing 6 shows the relevant changes to the implementation, which is hereafter referred to as **Kernel4**.

Listing 6. OpenCL-optimised hashing kernel function (Kernel4)

```
        uint tmp = mixVectorReminder(a,b,c,rem0,rem1, rem2, length);
        out[id] = tmp;
```

4. Results

In this section we provide a summary of results in Table 2, where we show execution times in milliseconds (ms) for each of our OpenCL implementations of the *lookup3* hashing function. From Table 2, it can be seen that the highest speedup relative to our naive OpenCL kernel **Kernel1** is achieved with **Kernel4** for the case of 1M keys, resulting in a $\sim$ 3.5x speedup. It should be stated at this point that we have implemented further lower-level performance optimisations, which result in up to $\sim$187x speedup relative to the naive OpenCL kernel, and which we are happy to discuss further via private communication.

While such optimizations are more low-level in nature, these are expected to become transparent to the user with upcoming

	Time[ms]		
Kernel (WGsize=4096)	**1M**	**2M**	**4M**
Kernel1 [2]	2092.25	4178.39	8384.41
Kernel2 [3.0.4]	1478.11	2957.99	5913.94
Kernel3 [3.0.4]	1077.01	2155.21	4304.54
Kernel4 [3.0.4]	596.20	1195.63	2362.02

Table 2. Execution time (ms) for each of the kernel designs describe above.

releases SDAccel, clearly demonstrating the huge potential of SDAccel as a means of empowering software developers to exploit FPGAs in the data centre much more rapidly than is currently possible with more cumbersone hardware-centric RTL flows.

5. Conclusions

We have reported on several implementations of an OpenCL-based design of a Bob Jenkins *lookup3* hashing function application on the Xilinx Virtex 7-based ADM-PCIE-7V3 platform using the SDAccel development environment. In particular, we have demonstrated how to best exploit the features of the OpenCL programming model available through SDAccel to improve performance by up to $\sim$3.5x relative to a naive OpenCL design of the application in question. It should be emphasised that nearly all of the performance gains described can be achieved through the high-level programming abstraction afforded by SDAccel's OpenCL development environment, with lower level optimisations expected to become transparent to the user soon. Such capability represents the potential for a huge step-change in productivity during the design phase of software defined solutions on heterogeneous platforms and, more generally, now provides organisations working in HPC and Technical Computing with a much lower-barrier-to-entry to exploiting the high performance, low energy capabilities of FPGA-based platforms.

Acknowledgments

The authors wish to thank Michaela Blott, Ken O'Brien and Kees Vissers of Xilinx for the great insight and assistance they provided during this investigation. The authors would also like to thank the Systems Team at ICHEC, especially Marco Grossi and Eoin McHugh for their invaluable support throughout.

References

[1] Memcached web page. http://memcached.org. 2

[2] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A hash table for line-rate data processing. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2):13:1–13:15, March 2015. 2

[3] Bob Jenkins. lookup3 code. http://burtleburtle.net/bob/c/lookup3.c. 2

[4] Xilinx SDAccel Developer Zone. http://www.xilinx.com/products/design-tools/software-zone/sdaccel.html. 2

[5] Xilinx. SDAccel Development Environment, User Guide, UG1023 (v2015.1) May 26, 2015. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1023-sdaccel-user-guide.pdf. 4, 5

[6] Khronos OpenCL Working Group. The OpenCL Specification, v1.0. https://www.khronos.org/registry/cl/specs/opencl-1.0.pdf. 4