# Pass a Pointer: Exploring Shared Virtual Memory Abstractions in OpenCL Tools for FPGAs

Felix Winterstein and George Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
South Kensington Campus, London, SW7 2AZ
Email: f.winterstein12@imperial.ac.uk

*Abstract*—**Heterogeneous CPU-FPGA systems are gaining momentum in the embedded systems sector and in the data center market. While the programming abstractions for implementing the data transfer between CPU and FPGA (and *vice versa*) that are available in today's commercial programming tools are well-suited for certain types of applications, the CPU-FPGA communication for applications that share complex pointer-based data structures between the CPU and FPGA remains difficult to implement. This paper focuses on programming environments providing a virtual memory space that is shared between the host CPU and one (or potentially several) FPGA devices. One example of shared virtual memory (SVM) is defined by the recent OpenCL 2.0 standard. SVM allows the software and hardware portion of a hybrid application to share complex data structures seamlessly (and concurrently) by simply passing a pointer, which greatly eases programming heterogeneous systems. We present a framework that automatically adds the physical infrastructure for SVM into a commercial OpenCL tool for FPGAs. This paper explores the design space for these building blocks and studies the performance impact. We show that, due to the ability of SVM-enabled implementations to avoid artificially sizing dynamic data structures and fetching data on-the-fly, up to 2x speed-up over an OpenCL design without SVM support can be achieved. Our framework is open-source and publicly available.**

## I. Introduction

Heterogeneous computing systems, coupling a central processing unit (CPU) with a field-programmable gate array (FPGA), such as Xilinx's Zynq [1] and Intel's Cyclone V [2] systems-on-chip (SoCs), Intel's forthcoming Xeon-FPGA hybrid [3], and IBM Power8 processors coupled with FPGAs [4] are gaining popularity in the embedded systems and data center market. This paper concerns the programming abstractions for these heterogeneous systems. High-level synthesis (HLS) promises a significant shortening of the FPGA design cycle. HLS flows based on OpenCL [5], a leading standard for heterogeneous computing, are especially gaining increasing popularity in the FPGA community and are promising candidates for a standardized programming environment suitable for CPU-FPGA hybrids.

The standard way to share data between the program on the CPU (referred to as the 'host') and the FPGA implementation (referred to as the 'device' and represented by one or more 'kernels') in today's FPGA OpenCL tools is to reserve a region in host system memory, which data is sent to and received from by the device, respectively. This method is compliant with the OpenCL 1.x specification [5]. One major drawback is that host and device use separate address spaces, implying that there is no guaranteed way to share pointers. If an application needs complex pointer-linked data structures, such as trees or linked lists, these must be converted to index-based data structures, for example by allocating a fixed-size array containing a tree and converting all pointers into indices into this array. This makes the sharing of such dynamically growing data structures between host and device complicated, especially when porting legacy code referencing large software libraries to heterogeneous architectures.

978-1-5090-4825-0/17/$31.00 © 2017 IEEE

The OpenCL 2.0 standard [6] addresses the complications due to separate address spaces by introducing *shared virtual memory* (SVM), an address space that is shared between host and device. In SVM, pointers assigned on the host can be seamlessly dereferenced on the device side and *vice versa* and address the same data in this case. Sharing dynamic data structures between the host and device can become as easy as passing a pointer, much like call-by-reference arguments in C/C++ function calls. This significantly raises the abstraction level and improves programmability for pointer-based programs on heterogeneous systems, which enhances the accessibility of FPGA-based heterogeneous computing to software developers. Furthermore, OpenCL 2.0 SVM introduces fine-grained host-device synchronization, which allows the host and device to access shared data structures concurrently and synchronize at the granularity of *atomic* load/store instructions. This enables true concurrency between software threads and hardware kernels in the presence of shared data structures. In this paper we ask the question: "What is the cost of including OpenCL SVM abstractions in FPGA-based heterogeneous computing?".

While the OpenCL 2.0 standard defines SVM [6], today's commercial FPGA-targeted OpenCL tools, as of writing, have SVM support only at internal beta stage and do not yet offer usable SVM functionality. Furthermore, as we shall see in Section V, a large design space for application-specific optimizations of the CPU-FPGA communication architecture underlying the SVM abstraction exists. This paper explores the design space and assesses the performance that can be expected from SVM programming abstractions. In the context of FPGAs, related work has either studied some of the required building blocks outside of an OpenCL environment [4], [7]–[11], or has evaluated parts of the OpenCL SVM specification on an experimental FPGA testbed for OpenCL

[12]. While the concept of our custom SVM building blocks is not new, we integrate them into an existing commercial OpenCL tool for FPGAs and provide an open research tool for SVM. Our framework automatically adds the physical infrastructure to enable SVM functionality in Intel's FPGA SDK for OpenCL [13]. This tool choice is exemplary; the same exploration could be performed with Xilinx SDAccel [14]. Secondly, this paper describes the details of our open-source framework.[1] The contributions of this paper are:

- ○ An implementation of a method enabling a shared virtual memory space for tightly coupled CPU-FPGA systems. The kernels on the FPGA device have access to the entire host system memory and can dereference any pointer in the host program, which is compliant with the OpenCL 2.0 *System SVM* specification. (Sections IV-A and IV-B)
- ○ An implementation of a method enabling fine-grained host-device synchronization through atomic load/store operations so as to allow host and device to access shared data structures simultaneously. (Section IV-C)
- ○ The integration of these methods as an add-on to the Intel FPGA SDK for OpenCL by automatically adding building blocks that enable SVM functionality. (Section IV-D)
- ○ A real-life benchmark implementation to demonstrate the improved programmability by using SVM. (Section III)
- ○ A design space exploration evaluating different implementations of the underlying communication primitives. We quantify the impact of the SVM abstraction in terms of resource utilization and execution time. On an Intel Cyclone V platform, we show that our SVM-based implementation is $1.5\times$ to $2.3\times$ faster than its counterpart following the normal OpenCL 1.x communication method. Depending on the data set, it also outperforms the special case when a shared physical memory is available on the device. (Section V)

## II. SVM Overview

The current generation of FPGA-targeted OpenCL tools is compliant with the OpenCL 1.0 standard. The data transfer between host and device in this specification is implemented as shown in Fig. 1 (left). The host accesses OpenCL buffer objects (Fig. 1 (left) shows an example with two buffers). These are mapped into the device address space with OpenCL-specific commands. In general, data transfer from host to the device and *vice versa* takes place before and after kernel execution, respectively. Synchronization between host and device only takes place at the `map`/`unmap` commands or at OpenCL events. OpenCL 2.0 brings additional data sharing and synchronization methods. It defines three types of SVM:

- ○ **Coarse-grain buffer SVM**: Shared data is placed in *OpenCL SVM buffer objects*. Sharing occurs at the granularity of regions (buffer objects). Host-device synchronization occurs at SVM buffer `map`/`unmap` commands. The only difference compared to regular OpenCL 1.x buffers is that host-side pointers are allowed in the SVM buffer.
- ○ **Fine-grained buffer SVM**: Shared data is placed in OpenCL SVM buffer objects. Sharing occurs at the granularity of regions (buffer objects). Host-device synchronization occurs at atomic load/store commands.

[1]Available at https://github.com/FelixWinterstein/FPGA-shared-mem

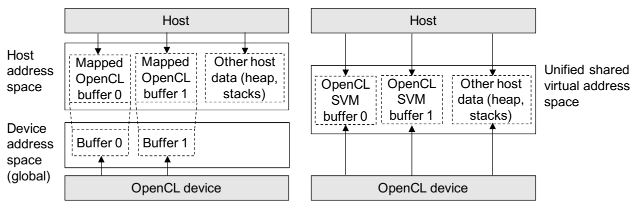


Fig. 1: Host-device communication in OpenCL 1.x (left); communication via OpenCL 2.0 fine-grained system SVM (right).

- ○ **Fine-grained system SVM**: The entire host address space is shared with the device. In addition to SVM buffers, the device can access all data in the system memory (heap, stacks). Host-device synchronization occurs at atomic loads/stores.

The host program usually runs on top of an operating system (OS) with virtual addressing, such as Linux. Hence, pointers in the host software carry 'virtual addresses' assigned by the OS. SVM ensures that host pointers are meaningful on the device. The first two modes require the explicit allocation of SVM buffers with the OpenCL `clSVMAlloc` function and, when the pointer to this buffer is passed to the kernel, it must be explicitly declared as an SVM pointer. In contrast, fine-grained system SVM provides the highest level of abstraction because each kernel on the device can access any pointer: those explicitly allocated with `clSVMAlloc` and those returned by the regular `malloc`/`new` functions. Fig. 1 (right) shows the fine-grained system SVM mode. This paper focuses on this mode, and specifically on the right-hand part of Fig. 1. The other modes can be easily implemented with the developed building blocks described in Section IV.

Because SVM accesses CPU system memory, access to cached host data is vital to good performance. Modern architectures enable processor cache coherency between CPU and FPGA through novel interfaces, such as the Intel QuickPath Interconnect (QPI) [3], IBM Coherent Accelerator Processor Interface (CAPI) [4] or the Accelerator Coherency Port (ACP) [2] in ARM-based SoCs. Our framework has been developed for the Cyclone V SoC, featuring cache coherency through ACP. In the future, it can be ported to Intel's Xeon-FPGA hybrid, once OpenCL support is available. The next section describes the advantages of fine-grained system SVM in the context of a representative example.

## III. Motivating Example

Tree structures are dynamic data structures commonly used in many applications. We consider a program that builds a 'kd-tree', a binary pointer-linked tree, and repeatedly traverses this tree to compute an output result. Our example is drawn from a high-performance implementation of a $K$-means clustering algorithm [15]. During the repeated tree traversals, significant computation is performed so that we implement the traversal phase on the FPGA and the sequential tree building phase (executed only once) in the host software.

Listing 1 shows the build-tree function. The `dataset` array is recursively split at index `n_lo` (Listing 1, Line 8); all data set entries below this index are assigned to the left child node and all entries above are assigned to the right

```
1 treeNode* buildTree(uint i, uint n, data_t *
      dataset) {
2   if (n <= 1) {
3     trNode* leafnode = new treeNode;
4     ...
5     *leafnode = ...;
6     return leafnode;
7   } else {
8     // split data set at index n_lo
9     ...
10    // recurse on children
11    treeNode* left = buildTree(i, n_lo, dataset);
12    ...
13    treeNode* right = buildTree(i+n_lo, n-n_lo,
          dataset);
14    ...
15    // set up parent node in post-order fashion
16    treeNode* parentnode = new treeNode;
17    parentnode->left = left;
18    parentnode->right = right;
19    return parentnode;
20  }
21 }
```

Listing 1: C-like pseudo code for a building a tree.

child, respectively (Lines 11 and 13). The final return value of the function is a pointer to the root node of the tree. This pointer is then passed to the FPGA device, which walks the tree starting from the root node dozens of times so as to compute the final result of the algorithm. The tree traversal implementation on the FPGA device is not shown.

This application benefits from SVM in two ways. Firstly, if we were to implement this application in OpenCL 1.x, we would have to modify the function `buildTree` and the surrounding host code to transform the pointer-linked tree structure into an index-based data structure. Listing 2 shows a way to do this. An array must be preallocated (`tree_array`, Listing 2, Line 1), which emulates the heap. Importantly, the size of the array must be known at compile time, which nullifies the advantages of dynamic memory allocation (artificially sizing dynamic data structures). All pointers must be converted into indices (type `uint` in this case). Calls to `new` are replaced by advancing a global index into the array (Listing 2, Lines 7-9 and Lines 23-25). It is important to note that, because heap memory is not freed in this function, this is a simplified method for replacing dynamic memory allocation in this example. It would be more complicated if the calls to `new` were interspersed with `delete` commands.

The above host code modifications may imply major refactoring of large code bases, especially when functions like `buildTree` are buried deep inside large legacy code libraries and have dependencies with the rest of the library. Hence, the complexity of code refactoring may become prohibitively large. On the other hand, fine-grained system SVM ensures that all pointers assigned in the host code can be dereferenced by the FPGA kernel, avoiding the need for host code refactoring altogether. It thus enables the seamless integration of FPGAs into existing software stacks by ensuring that only the portion ported to the FPGA has to be reimplemented and the surrounding code can remain untouched. Combined with modern high-level synthesis technology, we view this as a large step towards making FPGA-based computing more accessible to software developers.

```
1 uint buildTree(uint i, uint n, data_t *dataset,
      uint *index, treeNode *tree_array) {
2   if (n <= 1) {
3     treeNode leafnode;
4     ...
5     leafnode = ...;
6     // 'allocate' new array cell
7     *index = *index + 1;
8     array[*index] = leafnode;
9     return *index;
10  } else {
11    // split data set at index n_lo
12    ...
13    // recurse on children
14    uint left = buildTree(i, n_lo, dataset, index,
          tree_array);
15    ...
16    uint right = buildTree(i+n_lo, n-n_lo,dataset,
          index, tree_array);
17    ...
18    // set up parent node in post-order fashion
19    treeNode parentnode;
20    parentnode.left = left;
21    parentnode.right = right;
22    // 'allocate' new array cell
23    *index = *index + 1;
24    array[*index] = parentnode;
25    return *index;
26  }
27 }
```

Listing 2: Transformed version of Listing 1.

The second motivation behind using SVM in this implementation is related to performance. In OpenCL 1.x, the array `tree_array` is assigned to an OpenCL buffer object and generally copied into the global address space of the device memory prior to kernel execution. This means that the full-size tree, as determined at compile time, is copied over to the device memory. In the context of FPGA boards, this memory is usually a dynamic random access memory (DRAM) connected to the FPGA. The data transfer between host system and device memory takes time that must elapse before the kernel starts working. As we shall see in Section V, the overhead of transferring the bulk data structure can be significant. In our example application, however, the portion of the tree that is actually accessed by the kernel is much smaller than the full-size tree. We evaluate the percentage of the accessed tree nodes on synthetic data sets, which consist of $N = 2^{20}$ three-dimensional vectors to be clustered. The test vectors are distributed among 128 clusters following a normal distribution with varying standard deviation $\sigma$. As Fig. 2 shows, the percentage of visited, *i.e.* actually accessed, tree nodes depends on the value of $\sigma$. We also add the well-known Lena image benchmark data to include a real-life test set. Without going into the details of the implementation, Fig. 2 shows that the FPGA kernel will access only $5\%$ to $28\%$ of the tree, and only $12\%$ in realistic scenarios.

We argue that similar characteristics are commonplace in many other 'irregular' applications, such as other tree-based algorithms, which make data-dependent accesses to a data structure. Hence, transferring the full-size tree causes unnecessary data transfer. SVM avoids the explicit copy of the bulk data into the device memory ('zero-copy') and allows kernels to fetch data from system memory on-the-fly.

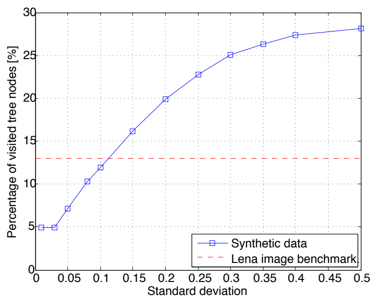


Fig. 2: Percentage of visited tree nodes relative to the full tree.

In summary, fine-grained system SVM eases programmability by avoiding extensive host code refactoring when porting applications to heterogeneous systems. In some cases, such as in applications with a varying, non-deterministic memory footprint with an average size smaller than the worst-case size, SVM can deliver performance improvements by allowing kernels to transfer data on-the-fly. In order to enable SVM functionality in OpenCL implementations, several building blocks on the FPGA device and the host side must be added and instantiated during the compilation process. The next section describes our implementation of these building blocks and their integration into the Intel FPGA SDK for OpenCL.

## IV. Custom SVM Infrastructure

OpenCL fine-grained system SVM requires three baseline functions. Firstly, host pointers carry virtual addresses assigned by the OS through the system calls `malloc` and `new`. However, memory accesses made through the kernel bus interfaces at register transfer level (RTL) use physical addresses. Consequently, a translation from virtual to physical addresses is required to make host pointers meaningful in the device address space. Secondly, the virtual-to-physical address translation must be embedded in a memory management unit, which handles the SVM accesses by the kernel and performs the actual data transfer. Thirdly, the fine-grained host-device synchronization for concurrent memory access through atomic load/store operations requires a mechanism to ensure that a host or kernel access to shared data is guarded against an interfering access to the same location by the other side until the access has been completed (atomicity of the access).

The target platform of our OpenCL SVM framework is an Intel Cyclone V SoC, consisting of programmable logic and a dual-core 32-bit ARM processor running Linux and the Intel FPGA OpenCL Runtime Environment (Version 16.0). While our framework can be ported to different architectures in the future, some of the infrastructure details described below are specific to the Cyclone V architecture. Specifically, this applies for the virtual-to-physical address translation (due to the ARM-specific Virtual Memory System Architecture) and the physical bus interfaces connecting the ARM core with the FPGA (SoC-specific on-chip bus architecture).

### A. *Virtual-to-Physical Address Translation*

The host pointers passed to the kernel contain virtual memory addresses assigned by the OS. When the ARM core performs a memory access it derives the physical memory address from the virtual address by a look-up in the Linux *page table*. The page table resides in system

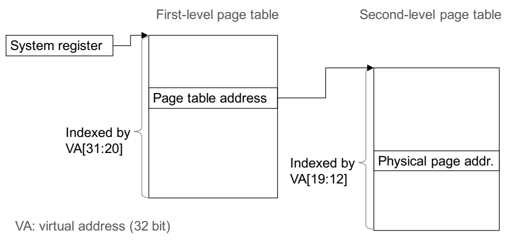


Fig. 3: Address translation using page tables in the ARM system architecture [16].

memory and stores the physical addresses of active memory pages. The ARM architecture uses a two-level translation table hierarchy as shown in Fig. 3. The base address of the first-level translation table is stored in a CPU system register (`TTBR0`) [16], which can be read out in Linux kernel mode. The first-level table is indexed by the upper 12 bits of the pointer value (32-bit virtual address), followed by a look-up to obtain the base address of the corresponding second-level page table. The latter is indexed by a second portion of the virtual address bits. The second look-up then results in the physical address of the 4 KB page. The physical memory address of the requested data is composed of the page address and the lower 12 bits of the virtual address.

In our framework the FPGA kernels are considered on equal terms with the CPU and shall have the capability to initiate accesses to CPU system memory themselves, independently of the host program. Therefore, the kernel must possess its own virtual-to-physical address translation implemented in FPGA logic. The translation procedure precedes every SVM access made by the kernel. The translation unit directly accesses the Linux page table in system memory via the main bus bridge connecting the ARM cores with the programmable logic through the ACP. In order to inform the kernel about the base address of the first-level page table, the value of the `TTBR0` system register is read out during the OpenCL initialization code in the host program (using a driver module provided by our framework) and is passed to the kernel as a default argument. With this base address, the translation unit performs two system memory reads so as to walk the first and second-level page tables and to obtain the physical address of the request word. In the rare event of page faults, *i.e.* virtual addresses for which no active physical page exists, these are detected during the address translation and trigger a CPU interrupt (using a spare FPGA-to-host interrupt request port in the Cyclone V). The software interrupt routine fetches the missing page, after which the address translation by the kernel is repeated. By default, page table walks are performed at the granularity of 32-bit words. The implementation of the address translation unit is part of the following section.

### B. *Memory Management Unit*

The memory management unit (MMU) integrates the virtual-to-physical address translation and the load/store functionality to perform the actual data access. The unit is an RTL module, which is added to the FPGA kernel logic. The address translation consists of two system memory reads, while the actual data access is either a read or a write access to the system memory. Fig. 4 shows the implementation

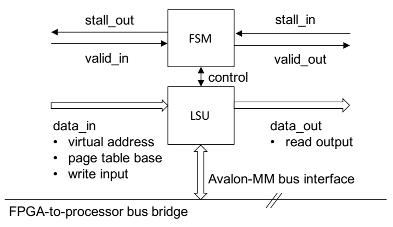


Fig. 4: Basic memory management unit.

of a basic memory management unit. Its interfaces with the rest of the kernel logic are FIFO-like Avalon streaming input and output interfaces, with signals indicating the availability of new data (`valid_in` and `valid_out`) and back-pressuring (`stall_out` and `stall_in`) as well as data inputs and outputs. The unit consists of a load/store unit (LSU), which handles the memory access requests to the system bus. The LSU connects to the FPGA-to-processor on-chip bus bridge (memory-mapped Avalon-MM interface) on the Cyclone V SoC in order to make cache coherent accesses to the CPU system memory through the ACP. The LSU contains a FIFO-buffer, which buffers the output data until it can be picked up by a downstream node. A finite state machine (FSM) controls the LSU and manages the stall and valid signals for the interaction with the rest of the kernel logic. In the basic version of the memory management unit, the two page table look-ups for address translation and the actual data accesses are time-multiplexed on a single LSU.

We explore several design choices for the memory management unit. Fig. 5 shows a pipelined version of the unit in Fig. 4. Instead of time-multiplexing the three bus accesses, it spatially unfolds the required LSUs and FSMs so that the kernel module accepts new requests as soon as the first pipeline stage is done and process multiple SVM requests in-flight. The arbitration of the concurrent accesses to the FPGA-to-processor bus bridge is offloaded to an Avalon bus interconnect. While the two versions trade throughput for FPGA area, we can configure the LSUs to include features that further influence this trade-off. Firstly, the way the LSU handles a sequence of requests can be altered:

- ◦ **Basic**: The LSU processes one request at a time. It stalls the upstream node until the memory request has been completed.
- ◦ **Static burst coalescing**: If the addresses in a sequence of $n$ requests are known to be strictly consecutive, these are compiled to a burst access of length $n$ (assuming $n$ is less than the maximum burst length), which is generally more efficient than a sequence of individual accesses. The LSU stalls the upstream node until the burst access has been completed.
- ◦ **Dynamic burst coalescing**: If the addresses in a sequence of $n$ requests are not known to be always strictly consecutive, but are consecutive in most of the cases, dynamic burst coalescing can be efficient. As long as the arriving request has a consecutive address with respect to the previous requests, the LSU keeps collecting requests before issuing them to the bus in a burst and aims to compile the largest possible burst.

Secondly, the individual LSUs can be configured to include a direct-mapped on-chip cache. In the case of the LSUs for the virtual-to-physical translation (`LSU 0` and `LSU 1` in Fig. 5, the cache corresponds to a *translation lookaside buffer* (TLB),

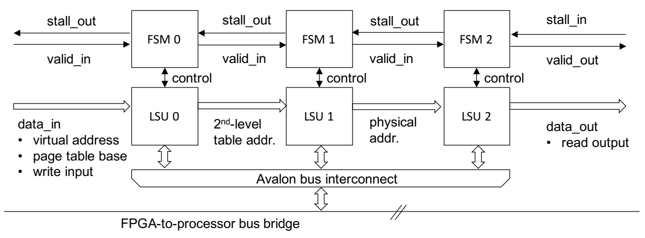


Fig. 5: Pipelined memory management unit.

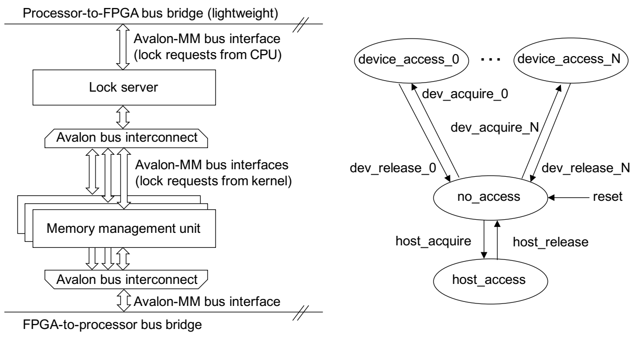


Fig. 6: Lock service architecture (left) and state machine inside the lock server (right).

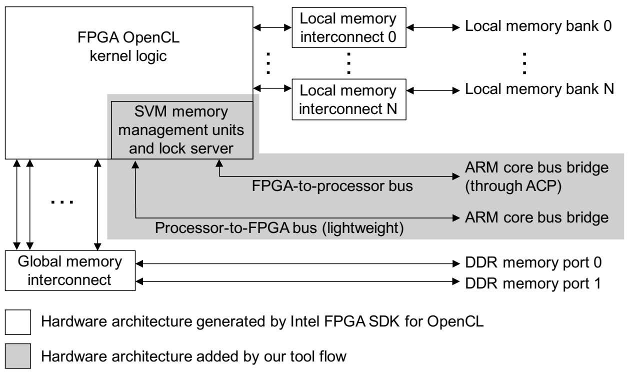
which stores the recent history of page table look-ups.

An FPGA kernel generally contains as many memory management units as SVM load/store instructions in the OpenCL code. All units connect to the FPGA-to-processor bus bridge through an Avalon bus interconnect. The following section describes an extension of the SVM infrastructure in order to enable fine-grained host-device synchronization.

### C. Atomic System Memory Accesses

In addition to supporting a shared address space, our memory management unit can be configured to include a host-device synchronization mechanism enabling atomic load/store operations. The mechanism is implemented using a *lock service* underneath the high-level view of atomic memory instructions provided to the programmer. To this end, the memory management unit has an additional Avalon bus interface to request ownership of a lock. Fig. 6 (left) shows the lock service architecture. At its core, a *lock server* serves the requests for acquiring and releasing the lock from both the host program and one or several MMUs in the FPGA kernel. The latter connect to the server via an Avalon bus interconnect. Lock request commands from the host side are received through another CPU-FPGA bus bridge in the Cyclone V architecture: the lightweight processor-to-FPGA bridge, which is specifically design for low throughput, low volume transfers.

Fig. 6 (right) shows the state machine inside the lock server. The commands `host_acquire`, `host_release`, `dev_acquire_x` and `dev_release_x` are buffered in a request queue and indicate the wish of the host and device to acquire or release the lock, respectively. The lock server acknowledges the successful completion of a request to the client. Memory management units stall until this request is received before issuing the actual memory operation. On the host side, an atomic memory operation polls memory mapped



FPGA OpenCL kernel logic
Local memory interconnect 0
Local memory bank 0
Local memory interconnect N
Local memory bank N
SVM memory management units and lock server
FPGA-to-processor bus
ARM core bus bridge (through ACP)
Processor-to-FPGA bus (lightweight)
ARM core bus bridge
Global memory interconnect
DDR memory port 0
DDR memory port 1
Hardware architecture generated by Intel FPGA SDK for OpenCL
Hardware architecture added by our tool flow

Fig. 7: Integration of the SVM building blocks into Intel FPGA OpenCL designs.

TABLE I: Reference data set for the design space exploration.

| Full tree size (number of nodes) | Nodes visited by kernel | Accessed portion |
|---:|---:|---:|
| 2097151 | 249899 | 11.9% |

processor-to-FPGA bus bridge while waiting for the acknowledgment. The lightweight processor-to-FPGA interface is also used to exchange data for page fault handling.

*D. Tool Integration*

Our SVM framework is implemented as an add-on to the Intel FPGA SDK for OpenCL. The SVM building blocks described above are added to the tool flow and are invoked by users in OpenCL host and kernel code. On the host side, an application programming interface (API) is provided, which allows the programmer to use and interface with our SVM functions. Our framework also provides an API for the kernel side. The programmer instantiates the SVM load/store instructions through functions calls in the kernel code. These calls trigger the instantiation of the custom RTL modules in the FPGA logic in addition to the regular kernel logic. Fig. 7 shows the hardware design generated by the OpenCL SDK. The gray-shaded parts in Fig. 7, namely the memory management units, the lock server, the additional Avalon bus configurations and the ACP configuration are automatically added to the OpenCL build flow. To this end, the OpenCL flow is halted after the generation of the system design and the generated RTL is modified by an automated source-to-source transformation so as to include the custom SVM functionality. The next section evaluates the performance of our SVM infrastructure and presents a design space exploration covering the different implementation options of the memory management units as described in Section IV-B.

## V. Evaluation

Raising the level of programming abstraction through SVM comes with a cost in terms of FPGA resources because additional hardware functions must be included in the design. It also has an impact on the performance in terms of execution time, which is either a performance gain or penalty, depending on the use case scenario and the implementation of the SVM building blocks. The following subsections quantify these two metrics using the Intel Cyclone V SoC as a target platform.

*A. Design Space Exploration*

Our open-source framework allows users to quickly explore different design options for the SVM infrastructure.

TABLE II: Parameter space.

| Parameter | Value range |
|---|---|
| MMU pipelining | non-pipelined / pipelined |
| LSU implementation | basic / static burst coalescing / dynamic burst coalescing |
| On-chip TLB size | 0; 128 Byte; 256 Byte; 2048 Byte; 4096 Byte |
| On-chip cache size | 0; 128 Byte; 256 Byte; 2048 Byte; 4096 Byte |

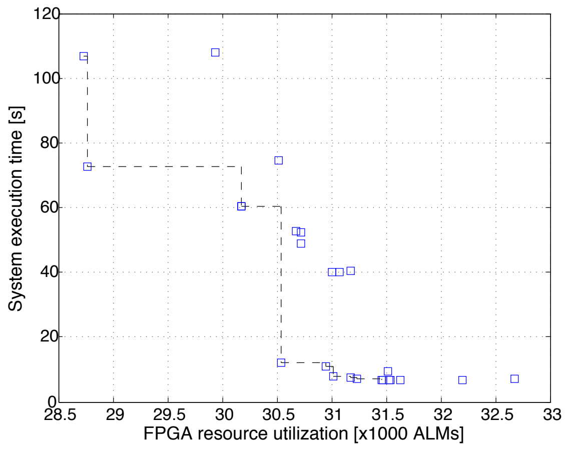


Fig. 8: Design space exploration for the SVM infrastructure.

The following design space exploration uses the clustering benchmark application from Section III, which shares a binary pointer-linked tree structure between host and FPGA device. We use a synthetic input data set of $N = 2^{20}$ three-dimensional vectors to be clustered. The test vectors are distributed among 128 clusters following a normal distribution with standard deviation $\sigma = 0.10$ as in Fig. 2 (Section III). The host builds the tree and passes the root pointer to the device. The fully pipelined kernel on the device accesses the tree nodes consisting of $14 \times 32$-bit words through different implementations of the SVM MMU. The rest of the kernel is the same in all designs. Table I shows the properties of the input data set. Table II shows the explored parameter space for the MMU implementation and Fig. 8 shows the execution time over the resource consumption of the whole FPGA design in terms of Adaptive Logic Modules (ALMs). The execution time is the *total system run time* including the tree build-up by the host software and the kernel run time. All results are taken from fully placed and routed FPGA designs executed on the Cyclone V SoC.

The results can be partitioned into three groups: non-pipelined MMUs (left hand side of the graph), pipelined MMUs with basic LSUs (center region) and pipelined MMUs with burst support (bottom right). We observe a significant span of execution time (more than $10\times$ speed-up from slowest to fastest design), which illustrates the importance of such a design space exploration. Overall, the full designs use up to 75.3% of the ALMs and up to 100.0% of the M10K RAMs available on the Cyclone V. The achievable clock rate varies only marginally between 102.7 and 114.1 MHz. The resource increase by including the SVM infrastructure varies from 4.4% to 11.2% of the ALMs and from 2.5% to 14.1% of the M10K RAMs, respectively, compared to a non-SVM OpenCL design. The fastest and most resource-intensive MMU implementations are pipelined and include dynamic burst coalescing LSUs supported by TLBs and a cache,

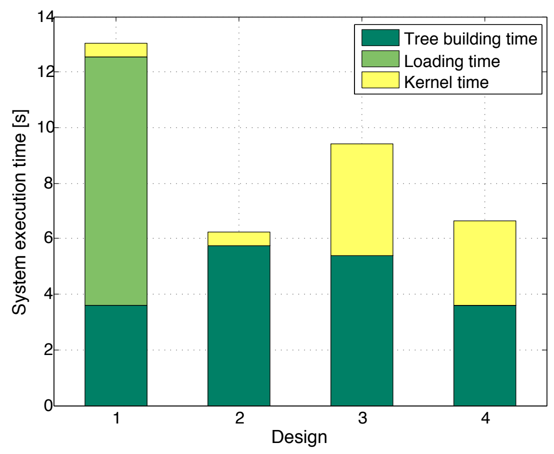


Fig. 9: Time break-down for non-SVM/SVM OpenCL designs.

respectively. We observe that small TLBs of 256 Bytes are sufficient to achieve a 99.4% hit rate so as to reduce memory accesses due to page table look-ups to a minimum.

## B. Comparison with OpenCL 1.x

We firstly compare the system execution time for OpenCL implementations of the benchmark application using the previous data set with and without our SVM infrastructure, analyzing the following design points:

1) **Standard OpenCL 1.x implementation**. This version does not include SVM functionality and uses the modified host code in Listing 2. The entire tree structure is placed in a traditional OpenCL buffer and copied into the device memory before kernel start.
2) **OpenCL 1.x implementation using physically shared memory**. This is the same setup as **Design 1**, but the the buffer containing the tree is allocated in a physically shared memory region. Hence, the copy from host memory to device memory can be avoided. We include this 'zero-copy' design as the lower bound for host-device communication time in OpenCL 1.x. Importantly, this design point is a special case because this mode is only available on systems which use the same memory chip for the CPU system memory and the FPGA memory (such as Intel Cyclone V or Xilinx Zynq).
3) **OpenCL design with static burst coalescing SVM MMU**. This is a pipelined MMU with TLBs and a data cache of 256 Bytes.
4) **OpenCL design with dynamic burst coalescing SVM MMU**. This is a pipelined MMU with TLBs and a data cache of 256 Bytes.

The pipelined kernel implementation is the same in all designs, except of the memory access units. The main differences in kernel time solely stem from different memory access times. Fig. 9 shows the system execution time break-down for the four designs. Due to the unavailability of direct memory access (DMA) between CPU and FPGA on the Cyclone V [13], **Design 1** spends a significant amount of time on copying the tree buffer over to the device memory ($> 8$ s). **Design 2** avoids the loading time altogether. Other platforms, without shared physical memory but with DMA support, will be at

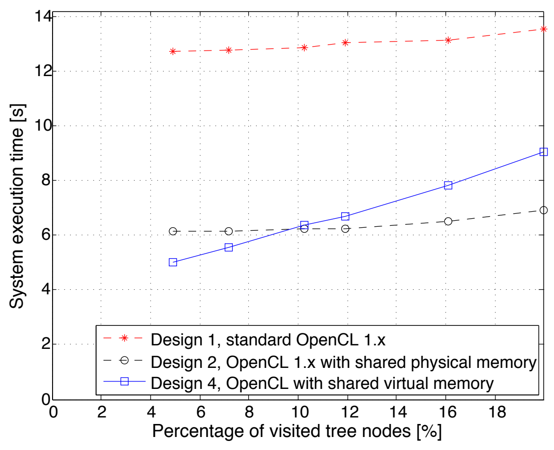


Fig. 10: Time comparison of non-SVM/SVM OpenCL designs.

a relative position between these two extremes. **Design 2** allocates the 'tree array' on the host side with the OpenCL command `clEnqueueMapBuffer` instead of the regular `new`/`malloc` commands, which results in slower host accesses to the tree during tree build-up. Both non-SVM kernels access the DDR memory directly, so kernel execution is fast. In contrast, the two SVM designs have longer kernel execution times due to slower accesses to the tree data, but they also avoid the initial copy of the whole tree. For **Design 3**, the 14 32-bit words stored at a tree node must be guaranteed to be at physically contiguous memory locations. This complicates the memory allocation for tree nodes, which introduces a time penalty for both the host and the devices memory accesses. **Design 4** dynamically aggregates bursts on-the-fly and does not strictly require this guarantee. The design can use the regular `new`/`malloc` commands for building the tree and has the best performance across all SVM designs. Our SVM implementation with a pipelined, dynamic burst-coalescing MMU in **Design 4** outperforms the standard OpenCL 1.x implementation by $2\times$ and achieves an execution time close to the OpenCL 1.x implementation with shared physical memory.

Next, we compare **Design 1**, **Design 2** and **Design 4** over a range of different input data sets. The value of $\sigma$, as described in Section 8, is varied from 0.01 to 0.20, which results in different percentages of the tree nodes accessed by the kernel relative to the full-size tree. This corresponds to a range of 5.5 MB to 22.3 MB of tree data transferred to the kernel. Fig. 10 shows that the SVM-enhanced design outperforms the standard OpenCL 1.x design over the whole range. If the kernel footprint in system memory is small (less than 10% of the full-size tree), our SVM implementation also outperforms the special case of OpenCL devices with shared physical memory.

## VI. Related Work

Much of the related work on hardware support for system memory accesses by FPGAs in heterogeneous systems has focused on implementations outside of OpenCL environments. Garcia and Compton [7] implement a method to use virtual addresses in programmable logic by caching recent virtual-to-physical address translations on the FPGA. The cache (TLB) is managed by system software. Misses in this cache, however, can cause considerable delays in their implementation. Our approach gives hardware kernels direct access to the system page tables and allows them to manage the address translation autonomously (except in the rare event of page faults).

Lange and Koch [8] present a lightweight address translation without software intervention by placing all system memory areas of a program inside a DMA buffer. As a result, virtual and physical addresses differ only by a fixed offset, which simplifies the translation. The approach requires modifications to the Linux kernel. Similar to OpenCL 1.x buffers, it requires *a priori* fixed sizing of the data, which introduces similar disadvantages as discussed in Section III. Their subsequent work in [9] addresses this issue by implementing logic to walk the CPU-managed page table in hardware, which is in line with our approach in Section IV. Agne *et al.* [10] use a similar approach as Lange and Koch. In extension, they focus on integrating the virtual memory management into the ReconOS [10] operating system and programming environment. Recent work by Cong *et al.* [11] presents an extensive study of application-specific optimizations of the TLB implementation in multi-accelerator scenarios so as to minimize the number of page table walks, which could be applied to FPGA accelerators. They propose a TLB hierarchy consisting of small per-accelerator TLBs and a large shared TLB. The small number of remaining TLB misses are offloaded as page walk requests to the host core MMU.

The underlying MMU principles in the above work are similar to the implementation in our framework and these works, in parts, explore more details of the MMU implementation. However, our main goal is to make this functionality accessible to users in the context of SVM support in a standard OpenCL programming environment and to compare its performance impact with implementations following the OpenCL 1.x standard, which is currently by state-of-the-art FPGA vendor tools. The work by Mirian and Chow [12] is notable in that it explores system SVM abstractions in the context of an OpenCL test framework consisting of embedded MicroBlaze softcores on an FPGA. They contrast host-based address translation with hardware-based MMUs and conclude that page table logic in the accelerator provides the fastest implementations. In contrast, we build on this insight and explore a design space for hardware-based MMUs. Furthermore, our open-source framework adds host-device synchronization mechanisms to the infrastructure and directly integrates into an industrial OpenCL SDK for FPGAs targeting standard hardware platforms.

## VII. Conclusion

OpenCL SVM significantly eases the programming of heterogeneous CPU-FPGA systems by giving FPGA kernels the ability to dereference pointers created in the host software seamlessly and by providing mechanisms to ensure data consistency in the case of concurrent sharing of data structures between host and device. We provide an open-source framework which automatically adds SVM functionality to implementations developed with the Intel FPGA SDK for OpenCL targeting a Cyclone V SoC. Using this framework, we present a design space exploration for the underlying SVM building blocks. With a real-life benchmark application, we show that the fastest variant of this design space exploration reduces the end-to-end system execution time by $1.5\times$ to $2.3\times$ compared to an equivalent OpenCL 1.x-based implementation. If the portion of accessed host data is small compared to the worst-case amount, our SVM implementation approximates the execution time of the special case using a shared physical memory. We conclude that, in addition to the ease of programmability, avoiding the requirement of artificially sizing dynamic shared data structures and fetching data only when needed, as enabled by OpenCL SVM, can result in performance gains in appropriate scenarios. Future work will include the exploration of a larger set of benchmarks, which we expect to exhibit different trade-offs for the underlying SVM building block implementations. We furthermore plan to deploy static program analyses to enable application-specific optimizations of these implementations.

## Acknowledgment

This work was supported by Intel Corporation, the EPSRC (Grants EP/P010040/1, EP/K034448/1), the Royal Academy of Engineering and Imagination Technologies.

## References

[1] Xilinx Inc., "Zynq-7000 All Programmable SoC Data Sheet: Overview," accessed: 2016-08-16. [Online]. Available: https://www.xilinx.com/

[2] Intel Corp., "Cyclone V Device Datasheet," accessed: 2016-08-02. [Online]. Available: https://www.altera.com/

[3] P. Gupta, "Accelerating Datacenter Workloads," accessed: 2016-09-30. [Online]. Available: http://www.fpl2016.org/

[4] J. Stuecheli, B. Blaner, C. R. Johns, and M. S. Siegel, "Capi: A coherent accelerator processor interface," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 7:1–7:7, Jan 2015.

[5] Khronos Group, "The OpenCL Specification Version: 1.2," accessed: 2016-02-12. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf/

[6] ——, "The OpenCL Specification Version: 2.0," accessed: 2016-02-12. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf/

[7] P. Garcia and K. Compton, "A Reconfigurable Hardware Interface for a Modern Computing System," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2007, pp. 73–84.

[8] H. Lange and A. Koch, "An Execution Model for Hardware/Software Compilation and its System-Level Realization," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2007, pp. 285–292.

[9] ——, "Low-latency high-bandwidth HW/SW communication in a virtual memory environment," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2008, pp. 281–286.

[10] A. Agne, M. Platzner, and E. Lubbers, "Memory Virtualization for Multithreaded Reconfigurable Hardware," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2011, pp. 185–188.

[11] J. Cong, Z. Fang, Y. Hao, and G. Reinman, "Supporting Address Translation for Accelerator-Centric Architectures," accessed: 2017-07-12. [Online]. Available: http://web.cs.ucla.edu/ haoyc/pdf/hpca17.pdf

[12] V. Mirian and P. Chow, "Evaluating shared virtual memory in an OpenCL framework for embedded systems on FPGAs," in *Proc. Int. Conf. on ReConFigurable Computing and FPGAs*, 2015, pp. 1–8.

[13] Intel Corp., "FPGA SDK for OpenCL Programming Guide," accessed: 2017-07-11. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_programming_guide.pdf

[14] Xilinx, Inc., "SDAccel Environment User Guide," accessed: 2017-10-02. [Online]. Available: https://www.xilinx.com/support/documentation-navigation/development-tools/software-development/sdaccel.html

[15] F. Winterstein, S. Bayliss, and G. Constantinides, "FPGA-Based K-means Clustering using Tree-Based Data Structures," in *Proc. Int. Conf. on Field Programmable Logic and Applications*, 2013, pp. 1–6.

[16] ARM, "ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition," accessed: 2016-09-29. [Online]. Available: http://infocenter.arm.com/