Offloading C++17 Parallel STL on System Shared Virtual Memory Platforms

Pekka Jääskeläinen[1,2][0000−0001−5707−8544], John Glossner[4,5], Martin Jambor[3], Aleksi Tervo[1][0000−0001−6050−7100], and Matti Rintala[1]

[1] Tampere University of Technology, Tampere, Finland
`pekka.jaaskelainen@tut.fi`
[2] Parmance, Tampere, Finland http://parmance.com
[3] SUSE, Prague, Czech Republic
[4] University of Science and Technology, Beijing, China
[5] Optimum Semiconductor Technologies, Tarrytown, NY, USA

Abstract. *Shared virtual memory* simplifies heterogeneous platform programming by enabling sharing of memory address pointers between heterogeneous devices in the platform. The most advanced implementations present a coherent view of memory to the programmer over the whole virtual address space of the process. From the point of view of data accesses, this *System SVM (SSVM)* enables the same programming paradigm in heterogeneous platforms as found in homogeneous platforms. C++ revision 17 adds its first features for explicit parallelism through its "Parallel Standard Template Library" (PSTL). This paper discusses the technical issues in offloading PSTL on heterogeneous platforms supporting SSVM and presents a working GCC-based proof-of-concept implementation. Initial benchmarking of the implementation on an AMD Carrizo platform shows speedups from *1.28X* to *12.78X* in comparison to host-only sequential STL execution.

Keywords: SVM, offloading, C++17, Parallel STL, HSA, GCC, heterogeneous platforms

1 Introduction

Heterogeneous computing is known for its potential in improving power efficiency in high performance and embedded computing [?,?]. However, utilizing heterogeneous platforms to accelerate general purpose applications is still hindered by programming difficulties. In comparison to parallel programming on homogeneous multicore CPUs, heterogeneous platforms, due to their distributed memory systems and multiple different *instruction set architectures (ISA)*, present additional challenges to the programmer including manual management of data transfers, explicit launching of kernels, and multiple dissimilar programming tools. It is evident that the programmer's productivity in terms of engineering time required to implement an optimized algorithm is related to the programmer's skill level in programming languages and tools. Therefore, to extend the utility of heterogeneous platforms, it is beneficial to avoid domain and platform specific tools and provide programming models familiar to a large number of existing programmers.

C++ is a popular general purpose programming language that is widely used, especially in high performance computing, system programming and the embedded world. In C++ revision 17, the language includes its first features for explicit parallelism referred to as the *Parallel Standard Template Library (PSTL)* [**?**]. While the PSTL is a welcome step towards utilizing parallel platforms programmed in C++ efficiently, C++17 does not yet provide a complete solution for additional issues in heterogeneous platforms such as non-uniform address spaces or explicit data transfers. The question of how much support C++ should actually provide, or whether direct heterogeneous platform support should be included in the standard, is still an active topic within the C++ standards committee.

Compute platform vendors have approached simplifying the programming of heterogeneous platforms by providing a *shared virtual memory (SVM)* that enables pointer sharing between the heterogeneous devices in the platform, but the degree of SVM support provided by the commercial platforms varies. However, the most advanced SVM implementations are now able to present a coherent programmer view over the whole virtual memory address space of the process. In these implementations, the complete virtual memory of the process is visible and coherent to all devices in the platform without explicit API calls. In this paper we refer to this level of SVM support as *system SVM or SSVM.* From the data access point of view, SSVM bridges the gap between heterogeneous and homogeneous platforms.

This paper discusses the remaining implementation challenges in heterogeneous automatic parallelization (i.e. offloading) of C++17 PSTL on SSVM heterogeneous platforms. The discussion is backed with a proof-of-concept GCC-based implementation[6] and its evaluation results on a commercial platform with SSVM capabilities.

The rest of the paper is organized as follows. Section **??** provides an overview of related technologies. Section **??** describes the C++17 standard and outlines its limitations and challenges in heterogeneous offloading. Section **??** reviews the concept of SVM in relation to C++. Section **??** describes a proof-of-concept (PoC) implementation on GCC, with initial benchmark results presented in Section **??**, finally, conclusions are presented in Section **??**.

2 Related Work

Features for supporting different degrees of SVM have appeared in various **lower level heterogeneous platform programming APIs** in the past few years. The *Open Computing Language (OpenCL)* standard from Khronos added an SVM API in OpenCL version 2.0 [**?**]. It defines multiple degrees of sharing and

[6] The implementation is being upstreamed to the GCC project. It will be published before the workshop takes place.

synchronization so that a conformant platform can be implemented. The optional *Fine-Grained System SVM* enables referring to any host memory without the need to allocate shared data objects using OpenCL-specific APIs and ensures data synchronization at atomic operation execution points. Thus, it fulfills the requirements of the "SSVM feature set" referred to in this paper. Khronos also specifies an intermediate language called SPIR-V [**?**], which together with fine grained system SVM provides an alternative implementation path for the platform abstraction components in the described proof-of-concept.

Heterogeneous System Architecture (HSA) [**?**] is a language neutral standard targeting heterogeneous systems. HSA specifies a coherent shared flat virtual memory as a core feature. Its "Full Profile" is similar to the *Fine-Grained System SVM* of OpenCL 2.0, allowing coherent sharing of data anywhere in the process address space. For the PoC described in this paper we used the HSA Runtime [**?**] as a heterogeneous platform middleware. The HSA Full Profile is ideal for this use primarily because its SSVM is designed to work seamlessly with the C++ memory model. There is also a wide selection of open source components that implement the different parts of the specifications, which can be used with the GCC toolchain. For example, its intermediate language HSAIL [**?**] has both frontend and backend support in the upstream GCC project.

NVIDIA is also moving in the direction of providing SSVM level capabilities. NVIDIA architectures since *Pascal* provide an advanced *unified memory* that provides SSVM level coherency and system-wide atomic operations [**?**,**?**]. A coherent view between the CPU and GPUs is implemented via on-demand virtual memory page migration. The first patch set (referred to as *heterogeneous memory management or HMM*) that provides support for this and various other types of SSVM implementations was upstreamed to the Linux kernel in version 4.14.

As PSTL is a recent addition to the C++ standard, there are only a few **open source implementations** of it available. Intel has published a PSTL implementation which is being upstreamed to libstdc++ of GCC [**?**]. It is implemented on top of the OpenMP 4's SIMD pragma and a parallel programming framework developed by Intel called *Threading Building Blocks (TBB).* TBB supports only homogeneous multicore CPUs, and offloading to heterogenous devices is not implemented.

SYCL ParallelSTL [**?**] implements PSTL on top of SYCL [**?**]. SYCL is a Khronos heterogeneous programming model for the C++ language. To the best of our knowledge, there is currently no open source implementation of SYCL available that can offload PSTL to heterogeneous devices. We also consider the HSA Runtime a more suitable implementation component for PSTL as it is tailored for middleware purposes and doesn't define its own higher level programming model which could just complicate the implementation in this case.

The closest related implementation to the proposed one is *HCC* [**?**], a heterogeneous compiler collection from AMD. Its earlier versions worked on top of a standard HSA Runtime and HSAIL, but it has now moved to direct to ISA compilation, making the device interfacing and the binaries the compiler produces less portable to other HSA platforms.

3 Heterogeneous Offloading of Parallel STL

Section 28.4. Parallel Algorithms of the C++17 standard is often referred to as "Parallel STL" or abbreviated as "PSTL". It describes an additional *execution policy* argument in the standard template library's algorithm API. Execution policies enable the programmer to declare that the algorithm library call, along with any user-defined functionality the call uses, is safe to execute in parallel. The user-defined functionality is collectively referred to as *element access functions (EAF)* in the standard. [**?**]

The following execution policies are supported: *sequenced_policy (seq)* forces serial execution of the EAFs in the calling thread of execution without interleaving, *parallel_policy (par)* promises parallelization safety of EAFs to multiple independent threads of execution, but does not guarantee "interleaving safety" *within* a single thread, and *parallel_unsequenced_policy (par_unseq)* additionally communicates interleaving safety in a single thread, e.g. by using SIMD instructions.

The parallelism related parts of C++17 focus on forward progress guarantees and their implications to parallelization safety on homogeneous processors. Execution on heterogeneous platforms, however, presents an additional consideration due to the different ISAs: The EAFs involved in the PSTL calls might rely on the target's (the host CPU's) properties, and assume the target specifics are uniform across all functions in the program. Thus, even if the user declares the strongest parallelization safety of *par_unseq*, which guarantees "vectorization safety", it is unclear if it is safe to offload to a device with a different ISA than in the host. Unsafe ISA features include different or undefined endianness, memory access alignment, and floating point rounding modes. However, an instruction-set abstraction layer such as HSAIL, which makes these attributes explicit, removes these ambiguities.

However, while it *might* be standard conformant to offload also *par_unseq* policy algorithm launches, offloading computation outside the host processor usually introduces additional invocation, synchronization and data transfer delays. Therefore, in the PoC presented in this paper, we added a new experimental execution policy we call *parallel_offload_policy* or *par_offload*. Using this policy, the programmer can declare the EAFs "heterogeneous offload" or "multiple-ISA" safe. *Par_offload* also implies *par_unseq* level SIMD-parallelization safety. An example code that calls the PSTL *transform* algorithm with this policy is shown in Fig. **??**.

4 System Shared Virtual Memory and C++17

The C++ memory model defines that data can be passed to other threads by pointers to objects in memory locations with unique addresses, thus implying a

```
std::transform(std::experimental::execution::par_offload,
               pixel_data.begin(), pixel_data.end(),
               pixel_data.begin(),
               [](char c) -> char {
                  return c * 16;
               });
```

Fig. 1. Example of offloading *transform* with the function to execute for each element defined as a lambda expression.

uniform address space. The pointers do not include the size of the pointed object. When considering heterogeneous offloading of PSTL, it is important to notice that the EAFs passed to PSTL algorithms are normal C/C++ functions, C++ function objects or "lambdas" which can refer to *any* data in the process. The accessed data is not limited to the containers handled by the launched algorithm. The EAFs can access arbitrary data allocated in any section of the process.

Having a unified coherent flat address space across all the processors in the heterogeneous platform, removes the major complexity of object access analysis that needs to track and wrap all the data accesses done by the EAFs, including their sizes. When reflecting against the C++ standard's memory model and the parallel algorithm requirements, a minimal feature set of a platform memory model that can semantically support PSTL algorithm offloading has the following properties:

– It presents a *flat process-wide address space* with unique physical locations identified by the pointer's address.
– All its locations can be *shared by default.* Data sharing between threads of execution does not need to be defined explicitly by the programmer.
– Shared locations can be updated with *atomic operations* without data races.

We rely on this SVM feature set in the described PoC and refer to it as SSVM (*system shared virtual memory*).

5 Proof of Concept Implementation

In order to identify the remaining challenges in heterogeneous offloading of C++17 PSTL when SSVM-level platform support is provided, we implemented a *proof-of-concept (PoC)* on top of GCC [**?**], the HSA runtime, and HSAIL.

We first describe an example of how an offloaded PSTL algorithm implementation looks like as an "HSA kernel". Fig. **??** presents a templated kernel implementation of the HSA-offloaded parts of the PSTL algorithm *transform*. *Transform* executes a user-defined function for all elements in an input container, writing the result to a corresponding position in an output container.

As can be seen in the example, a new function attribute type *hsa_kernel* is added for marking function definitions as offloaded functions that can be

```
template<typename ElementType, typename ResultType,
         typename FuncRetType, typename FuncArgType>
static void __attribute__((hsa_kernel))
__transform_array (void *args) {

  size_t i = __builtin_hsa_workitemflatabsid ();

  ResultType *d_first = *((ResultType**)args + 0);
  ElementType *first1 =  *((ElementType**)args + 1);

  d_first[i] =
    _FUNC_PLACEHOLDER <FuncRetType, FuncArgType> (first1[i]);
}
```

Fig. 2. Templated HSA kernel implementation of *transform* with an indirect call specialization placeholder.

launched from the host side. Other new attributes (not visible in the example) include *hsa_function* for functions that are compiled only to HSAIL and *hsa_universal* for functions which are compiled both to host native functions and HSAIL (later referred to as "universal functions").

Functions marked with the *hsa_kernel* attribute are *single program multiple data (SPMD)* style definitions called *kernels*. SPMD here means that the function defines what a single "work-item" (WI) does. The host runtime call (not visible in the example code) that launches the kernel defines the number of parallel work-items. In the case of *transform*, the host side launches the kernel with as many work-items as there are elements in the input container. The WI that executes the kernel function acquires its ID (location in the container) with a built-in call, reads the start location of the destination and the inputs from its argument data, and finally calls the user defined function.

Further technical details are discussed in the following subsections.

5.1 Binary Exchange Format

One of the issues in heterogeneous offloading is the program binary format. By the definition of *heterogeneous* computing, the potential target devices for the offloaded functions are diverse. Thus, the question of *binary portability* becomes an interesting one; even if storing a native binary for the host program, a portable heterogeneous binary format should preferably store an *intermediate language (IL)* or a virtual-ISA format for the offloaded kernels that can at finalization time be efficiently compiled to the final ISA. The time instant when to compile the IL to the real ISA can be delayed up until the program run time to enable maximum portability.

GCC's offload infrastructure uses an ELF-based "fat binary format", that, in addition to the ISA-targeted binary of the host program, can store different ILs, including HSAIL (which is used in the PoC), for the offloaded kernels. There is also an index for mapping the host-ISA and the IL versions of the included functions.

In the case of C++17, which doesn't support explicit marking of EAFs that might get offloaded, the compiler needs to somehow decide which functions should be compiled to the IL versions in the fat binary. Because the default assumption is that any function can be theoretically referred to in the EAFs, a full program link time call graph analysis starting from the PSTL algorithm calls would be needed to track all the possible functions recursively to collect the possible callees. For this first PoC, we decided to simplify this part and mark all functions as universal functions, without tracking whether they are potentially passed to PSTL algorithm calls or not. The bloat of the fat binaries produced by the duplication of functions is considered a lower priority problem.

It should be noted, however, that EAFs with certain unsupported features (such as exception handling or system calls) are dropped from the list of included IL functions based on the feasibility of their offloading. There are no inherent technical reasons to not support these features in EAF offloading and despite these omissions, the PoC is able to compile a large number of practical cases with the possibility to transparently extend the supported set in the future.

5.2 Indirect Calls and IL Specialization

While relying on SSVM takes care of pointers to data, the problem of multiple different ISAs and different disjoint *instruction* address spaces remain. Each targeted device can have its own ISA and therefore needs a separate program image in order to execute the offloaded EAFs. This means that there are potentially multiple copies of a function translated to the various ISAs in the heterogeneous platform simultaneously resident in the heterogeneous memory system. Furthermore, the targeted devices can have their own address spaces (ranges) for storing the functions; the flat host process address space can be assumed to contain only the bits for the host ISA versions of the functions. There is thus no uniform address range that can be used to uniquely identify a function's ISA binary. This complicates handling function pointers whose values can be treated as data in C/C++. While the sole efficient implementation of "universal function pointers" is an open problem, an additional consideration is the inefficiency of indirect calls in typical accelerator devices, thus they should be avoided for performance reasons whenever possible. Unfortunately, inheritance and virtual functions of C++ tend to increase their need.

In the PoC, indirect calls to the UEFs are supported via IL specialization. The PoC exploits the fact that the IL is compiled to the target ISA at runtime and at that point the called function is known. Thus, it is possible to "specialize" the IL version of the algorithm implementation at launch time by converting the indirect call to a direct one. The specialization idea is visible in the *transform* example of Fig. **??**. It calls *_FUNC_PLACEHOLDER*, which is a function declaration annotated with a fourth new attribute type *hsa_placeholder*. The

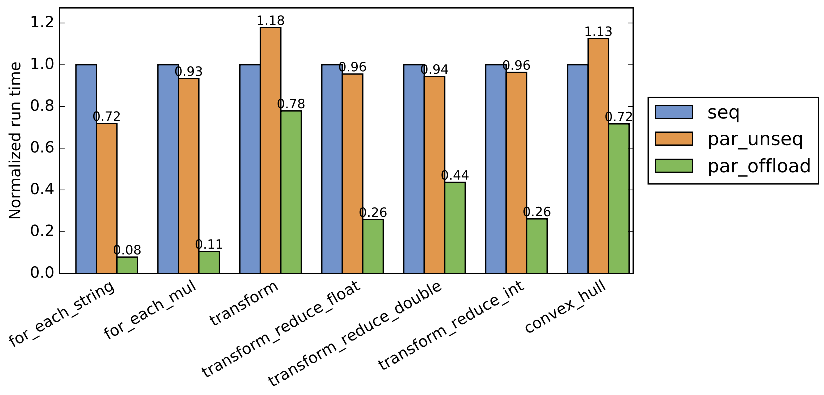

Fig. 3. Wall clock execution times, normalized to Intel PSTL seq execution policy.

placeholder is replaced by the concrete target function definition when the algorithm is launched and the used function is known. As this happens before the kernel is compiled to the targeted ISA, the specialization avoids indirect calls that launch the EAF inside the offloaded kernel, leading to enhanced optimization opportunities.

6 Evaluation

The PoC was integrated with the latest GCC development code base and benchmarked on an AMD Carrizo desktop PC. AMD Carrizo is a heterogeneous notebook range SoC that supports end user programmable SSVM across the CPU and the GPU via its HSA Full Profile support.

For evaluating the performance of the implementation, we created a benchmark set that exercises various PSTL algorithms that heavily call UEFs. The benchmarks were executed on an Ubuntu 16.04 based Linux system with minimal other system activity. The PoC (referred to as *par_offload*) was compared to the *seq* and *par_unseq* execution policies implemented by the latest version of the CPU-only PSTL contributed by Intel [**?**].

Each of the benchmarks was executed 10 times in a row, with the best runtime recorded. The input data for the benchmarks is reinitialized for each run in the CPU side to avoid the effect of data being migrated only in the first run to the GPU caches, which would give unrealistically optimistic results for the later runs. Thus, the measured run time includes the transfer of input data blocks from the main memory or the CPU's cache to the GPU's local cache. Reading them back for verification or further processing is not included.

The benchmark descriptions and the measured wall clock runtimes are listed in Table **??**. Fig. **??** illustrates the relative execution times normalized to Intel PSTL's sequential run time.

Table 1. The benchmarks and their runtimes. Wall clock time (μs) reported for Intel PSTL when launched with the **seq**uential execution policy. Other runtimes given as relative speedups in comparison to it.

	seq	par_unseq	par_offload
for_each_string			
Convert a 100 MB std::string to upper case	208303	1.39X	12.78X
for_each_mul			
Multiply 100MB of char data with a constant	156641	1.07X	9.48X
transform			
Gamma correct a 800x600 image	13800	0.85X	1.28X
transform_reduce_float			
Dot product of 100M element float vectors	1121486	1.05X	3.87X
transform_reduce_double			
Dot product of 100M element double vectors	2300442	1.06X	2.29X
transform_reduce_int			
Dot product of 100M element int vectors	1109145	1.04X	3.83X
convex_hull			
Convex hull from Intel PSTL examples	922030	0.89X	1.40X

The best **12.78X** speedup was received for *for_each_string*. The upper case conversion routine contains a bit of branching inside the EAF as it checks for the character ranges to avoid converting non-alphabetic characters, thus likely autovectorizes badly for *seq* and *par_unseq*, and executes more efficiently on the SIMT GPU. The 800x600 image in the *transform* gamma correction benchmark was added to serve as an example of offloading computation with relatively small input data to process. This case gets the most modest speedup of **1.28X**. The dot product cases received close to **4X** speedups with about half the performance for the double precision arithmetics as is expected.

The most apparent performance benefit of the single threaded *seq* policy is its possibility to compile the algorithm implementation together with the surrounding code at the call site. After template specialization and inlining the algorithm calls to the call site, autovectorization and other optimizations can be performed efficiently due to having more context information for the input and output iterators. In contrast, the *par_offload* policy requires a kernel dispatch call that isolates the algorithm implementation from its call site. The *par_unseq* policy suffers for the same reason due to its need to isolate the parallelized functionality to thread functions. Also, for the record, Intel's PSTL parallelizes much more efficiently on an Intel CPU due to TBB likely being better optimized for their own CPUs. We recorded *par_unseq* parallelization speedups up to 2.7X on a 4 HW thread Core i7 CPU.

7 Conclusions

SSVM is a key enabler for simplifying programming of heterogeneous platforms using traditional uniform address space languages such as C++. In this paper we discussed the remaining obstacles in seamless heterogeneous offloading of parallel standard template library algorithms introduced in the C++ revision 17 standard. With SSVM support in the platform, we identified efficient platform wide support for "universal function pointers" as a remaining key issue.

As a proof of concept, we presented technical details of a working heterogeneous offloading PSTL implementation implemented using GCC, the HSA runtime and the HSA Intermediate Language. Initial evaluation on an AMD Carrizo SoC in comparison to a sequential implementation running in the host CPU showed speedups from *1.28X* to *12.78X*.

For future work, we identified various further performance optimizations the PoC could benefit from. Utilizing HMM and the GCC's NVPTX backend to expand the set of supported platforms to new discrete GPUs from NVIDIA is also a very interesting future direction.

Acknowledgements

The authors would like to thank Academy of Finland (decision 297548) and the HSA Foundation for financially supporting the writing of this publication.