Mixed-Data-Model Heterogeneous Compilation and OpenMP Offloading*

Andreas Kurth
ETH Zurich
Integrated Systems Laboratory
Switzerland
akurth@iis.ee.ethz.ch

Koen Wolters
ETH Zurich
Switzerland
kwolters@ethz.ch

Björn Forsberg
ETH Zurich
Integrated Systems Laboratory
Switzerland
bjoernf@iis.ee.ethz.ch

Alessandro Capotondi
University of Modena
and Reggio Emilia
Department of Physics, Mathematics,
and Informatics
Italy
alessandro.capotondi@unimore.it

Andrea Marongiu
University of Modena
and Reggio Emilia
Department of Physics, Mathematics,
and Informatics
Italy
andrea.marongiu@unimore.it

Tobias Grosser
ETH Zurich
Scalable Parallel Computing
Laboratory
Switzerland
tobias.grosser@inf.ethz.ch

Luca Benini†
ETH Zurich
Integrated Systems Laboratory
Switzerland
lbenini@iis.ee.ethz.ch

Abstract

Heterogeneous computers combine a general-purpose *host* processor with domain-specific programmable many-core *accelerators*, uniting high versatility with high performance and energy efficiency. While the host manages ever-more application memory, accelerators are designed to work mainly on their local memory. This difference in addressed memory leads to a discrepancy between the optimal address width of the host and the accelerator. Today 64-bit host processors are commonplace, but few accelerators exceed 32-bit addressable local memory, a difference expected to increase with 128-bit hosts in the exascale era. Managing this discrepancy requires support for multiple *data models* in heterogeneous compilers. So far, compiler support for multiple data models has not been explored, which hampers the programmability of such systems and inhibits their adoption.

In this work, we perform the first exploration of the feasibility and performance of implementing a mixed-data-model heterogeneous system. To support this, we present and evaluate the first mixed-data-model compiler, supporting arbitrary address widths on host and accelerator. To hide the inherent complexity and to enable high programmer productivity, we implement transparent offloading on top of OpenMP. The proposed compiler techniques are implemented in LLVM and evaluated on a 64+32-bit heterogeneous SoC. Results on benchmarks from the PolyBench-ACC suite show that memory can be transparently shared between host and accelerator at overheads below 0.7 % compared to 32-bit-only execution, enabling mixed-data-model computers to execute at near-native performance.

Keywords Compilers, Heterogeneous Computer Architectures, Offloading, Memory Sharing, Data Models, Runtime Libraries, OpenMP

1 Introduction

Heterogeneous computers unite high versatility with high performance and energy efficiency by combining a general-purpose *host* processor with domain-specific programmable many-core *accelerator*s (PMCAs). The host manages input and output data as well as the application memory and *offloads* tasks that are highly parallel and/or domain-specific to one or multiple suitable accelerators [16, 38]. Due to the complexity of programming these systems, significant effort has been spent on developing programming models that retain high programmer productivity. A common way is to abstract the complexity through code annotations, indicating which code is to be offloaded, providing one unified code base. One de-facto standard programming model is OpenMP [34].

OpenMP 4.0+ [44] enables work to be offloaded from host to accelerators with the `target` directive and has been adopted for GPUs [2] and PMCAs in general [33]. Data is shared by copying from the *application memory*, which is

*This work was partially funded by the EU's H2020 project OPRECOMP (No. 732631), Polly Labs (Xilinx Inc, Facebook Inc, and ARM Holdings), and the Swiss National Science Foundation through the Ambizione program.
†Also with University of Bologna, Department of Electrical, Electronic, and Information Engineering.

managed by the host, to the *local memory* of the accelerator before the offload and back after the offload.

Application memory is growing rapidly: today 64-bit addresses are sufficient to handle data of hundreds of petabytes distributed over multiple nodes [55], but when multiple exabytes of data need to be addressed, 128-bit host processors will be required to manage application memory. Accelerators, on the other hand, are designed to work mainly on data in their local memory, which inherently grows at a lower rate than total application memory. The same trend can be observed in heterogeneous systems on chip (SoCs), where 64-bit hosts are common today, although accelerator memory is within the 32-bit addressable range [12].

This growing disproportion raises the question whether there is a fundamental need for accelerators to increase their *data width* solely to share pointers with the host. For each accelerator core, doubling the data width at least doubles the size of most of its components – the frontend, the register file, the arithmetic logic unit (ALU) (where the multiplier even grows quadratically), the load/store unit (LSU), and most internal buffers [63]. Furthermore, it usually doubles the longest combinatorial path, requiring at least one additional pipeline stage to prevent a reduction of the maximum frequency. For every executed accelerator instruction that does not fully exploit the wider data path (doubling SIMD parallelism), performance per area and efficiency of the accelerator effectively decreases. As with most other properties of accelerators [22], it is thus desirable to design the data width to match the needs of the target domain. To achieve this in the long term, as the application memory continues to grow, mixed-data-width systems are required.

The challenge in mixed-data-width systems is to transform offloaded pointers and types that have a data-model-dependent size from wide host values to narrower accelerator values while preserving their semantics and incurring as little run-time overhead as possible. While this could be done manually, doing so is error-prone and requires to rewrite existing libraries and applications. Therefore, heterogeneous compilers need to support multiple *data models* to bridge the disproportionate data widths in heterogeneous systems. However, to date, no heterogeneous compiler practically supports accelerators with a data model that differs from that of the host. Additionally, minimal hardware support to let accelerators access addresses outside their native data width has not been explored.

Contributions. In this work, we address these challenges. To our knowledge, this work is the first to:

1. Design and implement a mixed-data-model (64+32-bit) heterogeneous compiler, including full support for OpenMP offloading.
2. Discuss the challenges and options for implementing mixed-data-model compilation in current versions of the two main compilers, GCC and LLVM.

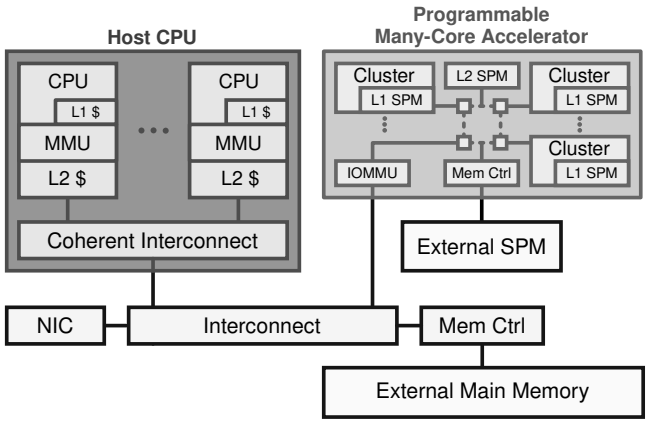

Figure 1. Architectural template of heterogeneous computers targeted by this work.

3. Discuss novel hardware options for extended addressing and implement and evaluate a minimal, non-intrusive option that does not require modification of any core or ISA.

Outline. This paper is structured as follows: After introducing the relevant background concepts in § 2, we explore the solution space to mixed-data-model OpenMP offloading in § 3, present our compiler solution in § 4, and describe the minimal accelerator hardware support for extended addresses in § 5. We show that our solution allows a 32-bit accelerator to transparently share memory with a 64-bit host at overheads below 0.7 % on PolyBench-ACC kernels in § 6. We compare to related work in § 7 and conclude in § 8.

2 Background

In this section, we introduce the heterogeneous compute and memory architecture targeted by this work (§ 2.1), OpenMP (§ 2.2), and data models (§ 2.3), and we discuss the state-of-the-art in offloading (§ 2.4) and heterogeneous compilation (§ 2.5).

2.1 Target Architecture

Fig. 1 shows the architectural template of heterogeneous computers we target in this work. The general-purpose host CPU is coupled to one or multiple PMCAs via an interconnect over which they share the external main memory and I/O peripherals, such as the network interface controller (NIC). The host CPU consists of one or more general-purpose application-class processing cores and has a memory hierarchy of virtually-addressed caches. The PMCAs consist of many minimal, domain-specific processing elements (PEs), potentially grouped in clusters, have a memory hierarchy of physically-addressed, software-managed scratchpad memorys (SPMs), and include an input/output memory management unit (IOMMU) to share the virtual memory space with the host. Host and PMCAs may implement different instruction set architectures (ISAs). There are

Data model	Width (in bits) of		
	`int`	`long`	**pointers**
ILP32	32	32	**32**
LLP64	32	32	**64**
LP64	32	64	**64**

Table 1. 32- and 64-bit data models common today.

many examples of such architectures in products ranging from high-performance computing (HPC) [24, 38] over high-performance SoCs [16] to low-power SoCs [17, 52] as well as in research [9, 21, 28, 59].

2.2 OpenMP and Offloading

OpenMP [44] defines a target-agnostic API based on preprocessor directives that are translated by the compiler into calls to runtime library (RTL) functions. Since version 4.0, OpenMP supports *offloading* of computation to accelerators with the `target` directive and data sharing through the `map` directive. The `target` directive determines which code is compiled for the host, the accelerator, or both. GCC and LLVM implement this *heterogeneous compilation* in very different ways (§ 2.5), and we focus on how this impacts handling different data widths of host and accelerator.

The `map` clause of the `target` directive specifies data to be shared for each offloaded kernel. OpenMP's data sharing model is copy-based: The host copies data from its virtual memory space to a physically-contiguous memory section, which accelerators can access without participating in the virtual memory system of the host. This restricts `map` to data structures that do not contain pointers. However, extensions for shared virtual memory (SVM) have been proposed and implemented [29, 33, 58]. That generalized variant of `map` effectively reduces offloading to passing pointers to shared data to the accelerator.

Such true pointer sharing is essential for three aspects: First, it eliminates one level of copying (from the host to the device memory space (still DRAM) and back). Second, it allows the accelerator to transfer only the data it requires directly to its closest memory level. Third, it enables offloading of pointer-based data structures. OpenMP 5.0 introduced the `requires` directive with the associated `unified_shared_memory` clause, which provides these pointer sharing semantics and makes `map` clauses on `target` constructs optional.

Our work supports both data-copy and pointer-passing offloading. When only copy-based offloading is required, simpler solutions could be found because the physically-contiguous memory section of the accelerator must inherently be addressable by the 32-bit accelerator.

2.3 Data Models

A *data model* defines the width of pointers and integer types that have a platform-dependent width. Table 1 lists 32- and 64-bit data models common today. In this work, we focus on pointers and discuss the challenges of offloading from a host with one data model to an accelerator with another in § 3.

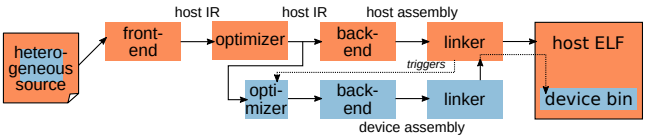

Figure 2. GCC implementation of OpenMP offloading. Red parts pertain to host compilation (top), blue parts pertain to accelerator/device compilaton (bottom). For device compilation, only a subset of optimization passes get executed.

2.4 Accelerator Address Space Restricted Offloading

There are already computers where accelerators have a narrower address width than the host [9, 17, 23, 48]. To share addresses between host and accelerators on such computers without compiler support, different fallback options are being used. All these options restrict the address space of user-space applications on the host while keeping the operating system (OS) in the native address space. First, host applications could be compiled for a different ISA that has a smaller address width but is compatible with the host ISA. For example, 64-bit ARMv8-A cores are user-space compatible with the 32-bit ARMv7-A ISA [3] and RV64 cores optionally implement an RV32 mode [60]. However, this is not possible for all ISAs. Second, host applications could be compiled for a different data model that has a smaller address width. For example, Intel introduced x32 for x86-64 [31]. A major drawback of this option is that it requires changes to the compiler, the standard library, and the kernel, which are relatively complex to maintain for the limited benefits it offers [32]. Third, some OSes, such as Linux, support restricting stack and heap addresses to a subset of the address space [25]. However, none of these fallback options allow host applications to use the full 64-bit address space, so they do not solve the problem we address.

2.5 Heterogeneous Compilation: State of the Art

GCC separates host and accelerator compiler to compile an application with OpenMP offloading, as shown in Fig. 2. The host compiler drives the compilation of a heterogeneous application and first lowers the source code to the GIMPLE intermediate representation (IR). When the host compiler finds a `target` section, it creates a new outlined function. Next, in the expansion phase, the host compiler replaces OpenMP directives with calls to functions in the host `libgomp` RTL. Finally, after optimizing the GIMPLE IR and as part of link-time optimization (LTO), the device compiler is invoked to transform GIMPLE IR to accelerator machine code [6].

In contrast to GCC, LLVM can natively compile for different targets and implements heterogeneous compilation as two separate compilations, as shown in Fig. 3. The necessary infrastructure was first presented in [2], the key feature being that the device compilation is largely independent from

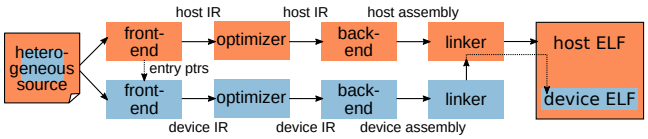

Figure 3. LLVM implementation of OpenMP offloading. Red parts pertain to host compilation (top), blue parts pertain to accelerator/device compilaton (bottom).

the host compilation, except for two points. First, the host compiler is responsible for annotating the parts of the source code that are visible to the accelerator, and forwards this information to the device toolchain. Importantly, this is done before the host has assumed any data model upon the code. Second, the host link action depends on the completion of the accelerator compilation, such that the accelerator code can be linked into a fat host executable and linkable format (ELF) file. We will discuss the impact of the different approaches of GCC and LLVM in § 4.1.

3 Mixed-Data-Model Offloading

The central problem in mixed-data-model offloading is to overcome the difference in the width of pointers between host and accelerator. As a practical example, consider sharing 64-bit pointers from an LP64 host with an ILP32 device. The device's ILP32 data model defines pointers to be 32 bit wide, so host pointers cannot be used as function arguments on the device. The device ISA defines memory access instructions on 32-bit registers (and potentially immediates), but provides no way to access 64-bit addresses. Thus, even though data values wider than 32 bit can be shared with 32-bit devices, they cannot be used as pointers or, from a lower-level perspective, as memory addresses.

To access addresses wider than the native width, the device needs to provide minimal hardware support, which the compiler can use through builtin functions. We will describe the hardware implementation of these functions in § 5; but for now we assume there are two runtime functions the wide-address load `wide_load(uint64_t wideaddr)` and the wide-address store `wide_store(uint64_t wideaddr)` that take fixed-width integers (e.g., `uint64_t` in C), wide enough to represent the host pointers, as arguments and perform the load from or store to the given address.

3.1 Mixed-Data-Model *OpenMP* Offloading

To use host pointers as arguments to the extended load and store functions, they need to be converted to fixed-width integers in all OpenMP `target` code and the `map` clause. In C, this could be achieved by replacing all host pointer types with `uint`*N*`_t` and all reading or writing dereferences with calls to the load or store function, respectively. In C++, this could be achieved by defining a class that wraps a host pointer and overloading its dereference and assignment operator. When this transformation is left to the programmer, it is highly intrusive and requires changes to applications and libraries, which opposes the goal of transparent offloading. Moreover, this transformation is incompatible with copy-based OpenMP offloading for arrays because the array dimensions are stripped from the `map` argument.

The concept of passing host pointers as fixed-width integers and replacing their use in device code with calls to functions is nonetheless valid, but the transformation has to be performed by the compiler.

4 Mixed-Data-Model Compilation

In this section, we discuss options to implement mixed-data-model compilation based on the concept presented in § 3 in GCC and LLVM.

4.1 Feasibility in GCC and LLVM

GCC lowers the source code of host and accelerator to the same IR, which is determined by the host compiler (details in § 2.5). This implies that the data model of the host must be used also for the device, and since GCC treats all pointers uniform in this respect, that the data model of the host defines the width of all pointers. Unless GCC's approach to heterogeneous compilation is changed fundamentally and the GIMPLE IR can represent pointers of different width, mixed-data-model compilation is infeasible in GCC.

LLVM, on the other hand, separates compilation for host and accelerator as much as possible (details in § 2.5) including the use of multiple device-specific IR modules. Also, LLVM supports different *address spaces*, each of which can have its own width, and allows to assign pointers to address spaces. Address spaces are defined in the *data layout* string, and each heterogeneous target architecture can define its own data layout. This makes LLVM a natural choice for our mixed-data-model compiler.

4.2 Front-end or Optimizer?

The first question to address is where to implement the transformation of pointers. The choice is between front-end or optimizer, because by the time the code reaches the back-end, it has to be reduced to types and operations that the target supports natively. In the **front-end**, the transformation would traverse the abstract syntax tree (AST) of the application. It would identify each host pointer that is offloaded to the device, replace its type with a fixed-width one, and replace its use with a function call. The main drawbacks of this option are that matching all relevant patterns in the AST is difficult and that it has to be implemented specifically for each language that is to be supported. In the **optimizer**, the transformation would operate on the IR of the compiler. LLVM's IR is in static single assignment (SSA) form, which allows for use-def chain traversals that are natural for replacing a pointer and all its uses. Also, the IR format is independent of the source language, so this option can be generalized to any language supported by the compiler. For

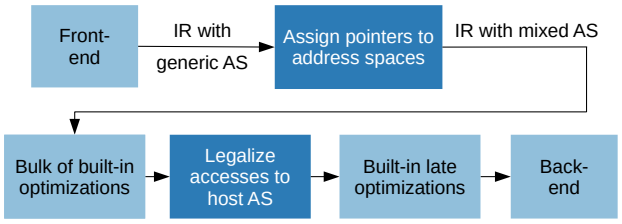

Figure 4. Flow chart of the extended (dark blocks) device compiler mid-end / optimizer passes.

these reasons, we implement our mixed-data-model compiler through optimizer passes.

4.3 Our Mixed-Data-Model Compiler in LLVM

To keep track of which pointers address values in the host and the accelerator memory, we assign them to separate host and accelerator *address spaces* (ASes). We refer to the accelerator AS as *device* AS to be aligned with common terminology. The *generic* AS defines the AS of pointers that are not explicitly assigned to an AS.

As shown in Fig. 4, we extend the compiler mid-end for the accelerator with two passes. The first pass assigns pointers from the generic AS to the host or the device AS. The second pass converts host pointers to fixed-width integers used to call the `wide_load()` and `wide_store()` builtin functions. The passes will be described in § 4.3.2 and § 4.3.3 respectively. First, we will select which AS to use as *generic* AS for the accelerator.

4.3.1 Choosing the generic address space

As host and device compilation are separate in LLVM, the compiler assigns the native AS as the generic AS for compilation. However, using the native accelerator AS as generic AS leads to problems when the device also has to handle wider-than-native pointers: dereferencing a host pointer that is not assigned to the host AS on the device uses only the lower bytes of the address, which results in an illegal memory accesses. To ensure correctness, the compiler must guarantee that each pointer is assigned to an AS that is wide enough to cover the full addressable memory.

Compiler assignment of pointers to an AS is simple in many cases, e.g., pointers used in OpenMP offloading (`map` clause) are always in the host AS. In full generality, however, use-def chains can trace to a load from memory, where potential aliases make the identification of all possible values, and thus the decision between host and device AS, a difficult problem. To avoid this pitfall, we argue that the native device AS is no longer a suitable generic AS when considering mixed-data-width compilation.

We therefore switch to using the host AS as generic AS also on the device. As host pointers are assumed to be wider than accelerator pointers, the host AS is a superset of the device AS. This gives trivial guarantees for correctness as the generic AS is wide enough to represent any pointer. Having the fundamental correctness guarantees, the compiler can *optimize* for performance by assigning pointers to the device AS when it is guaranteed to preserve correctness.

For this reason, we use the host AS as generic AS and optimize as many pointers as possible into the device AS.

4.3.2 Assigning pointers to the device address space

Our solution for assigning pointers to address spaces has two stages. First, we introduce a `__device` qualifier and let Clang expand that to an `__attribute(address_space)` corresponding to the device AS, which propagates the AS assignment to the IR. The intention of this qualifier is that developers of accelerator libraries (e.g., device stdlib and OpenMP RTL) use it on pointer arguments and return values to give the compiler *anchor points* for device pointers. For instance, a `malloc` in accelerator SPM will return a device pointer, and data transfers to the local accelerator SPM will take a device pointer as `dst` argument. All stack allocations in the device code are automatically in the device AS. The second step is to propagate the AS from the *anchor points* during compilation, such that the burden is not put on the programmer.

Algorithm 1 Assign pointers to device AS if possible.

```
1:  EntryPtrs ← {}
2:  for each module M do
3:      for each alloca A ∈ M do
4:          if A allocates a pointer of depth one then
5:              append A to EntryPtrs
6:          end if
7:      end for
8:  end for
9:  for each pointer P ∈ EntryPtrs do
10:     if HoldsOnlyDeviceASPointer(P) then
11:         NewP ← alloca of P in device AS
12:         ReplacePtr(P, NewP)
13:     end if
14: end for
```

The AS assignments are propagated through pointers that only have dependencies to the known *anchor points*. For all pointers passed directly through use-def chains, LLVM does this implicitly. For pointers passed via memory (e.g., explicitly in C as pass-by-reference, when invoking OpenMP kernels, or due to a transformation), the `AddressSpaceAssigner` pass, shown in Algorithm 1, ensures that the AS is properly propagated. We subsequently call the pointer stored in memory *inner pointer* and the pointer to it *outer pointer*. That is, *the outer pointer is used to pass the inner pointer via memory*. `AddressSpaceAssigner` first adds all stack-allocated outer pointers that store an inner pointer to the *EntryPtrs* set (lines 1 to 8). On lines 9 to 14, the pass checks if each outer pointer $P \in$ *EntryPtrs* only ever holds

device-AS inner pointers (Algorithm 2). If it does, it replaces the generic-AS inner pointer in P with a device-AS inner pointer in a new outer pointer $NewP$, and recursively replaces all uses of P with $NewP$ (Algorithm 3).

Algorithm 2 Determine whether an outer pointer P always holds an inner pointer in the device AS.

```
1:  function HoldsOnlyDeviceASPointer(pointer P)
2:      ▷ Check if each use U does not set the inner pointer of P outside
        the device AS.
3:      for each use U of P do
4:          if U is ptrtoint or load from P then
5:              continue
6:          end if
7:          if U is cast or getelementptr then
8:              ▷ Casts and GEPs return a new pointer to recurse on.
9:              if HoldsOnlyDeviceASPointer(U) then
10:                 continue
11:             end if
12:         end if
13:         ▷ Ensure that each inner pointer ever stored to P is in
            device AS.
14:         if U is store S to P and value of S is
              addrspacecast from device AS then
15:             continue
16:         end if
17:         ▷ Ensure that each function called with argument P does
            not modify the inner pointer of P.
18:         if U is direct call to function F and P is used
              read-only by F then
19:             continue
20:         end if
21:         ▷ If no match, inner pointer of P could be outside device
            AS.
22:         return false
23:     end for
24:     return true
25: end function
```

`HoldsOnlyDeviceASPointer` in Algorithm 2 determines whether an outer pointer P always holds a device-AS inner pointer. For this, the pass matches all uses of P against conditions known to not change the AS of the inner pointer. The list of conditions constitutes the bulk of Algorithm 2, but may not be complete. Since this is an optimization pass, the gist is to preserve correctness: If the conditions in the algorithm do not ensure that a pointer cannot be assigned a value outside the device AS, the AS migration is aborted on line 22. Additional conditions that preserve correctness could be added to further improve the optimization, but their omission does not compromise the correctness of Algorithm 2.

`ReplacePtr` in Algorithm 3 replaces each use of a pointer OP with a new pointer NP. We use it on line 12 of Algorithm 1 to replace host-AS pointers with device-AS pointers.

Algorithm 3 Replace all uses of a pointer.

```
1:  function ReplacePtr(old pointer OP, new pointer NP)
2:      for each use U of OP do
3:          if can replace OP by NP in U then
4:              NU ← clone of U with OP replaced by NP
5:              if type of NU ≠ type of U then
6:                  ReplacePtr(U, NU)
7:              end if
8:          else
9:              CI ← NP addrspacecast to type of OP
10:             replace use of OP in U with CI
11:         end if
12:     end for
13:     replace OP with NP
14: end function
```

For each instruction U that depends on the old pointer OP, the algorithm first checks whether the use of OP in U can be replaced with the given new pointer NP (line 3). Such a replacement is not possible, e.g., if U is a call to an external function: Because LLVM IR is strongly typed – including ASes – the AS of every use of the argument within the external function would have to be changed, which is not possible if the function is not visible to the compiler. In this case, `ReplacePtr` casts the resulting pointer back to the original AS (lines 9 to 10), to remain compatible. If OP can be replaced (lines 4 to 7), U is cloned into NU, which uses the new pointer NP instead of OP. For dereference chains, `ReplacePtr` needs to replace the entire chain with the new AS. Due to strong typing, instructions that return pointers have their type changed when the AS is modified. This is detected on line 5 and triggers a recursive call on line 6. The base case occurs when the type of NU is the same as the type for U, i.e., the instruction returns a non-pointer value. Once OP has been replaced in all uses, the original pointer OP is replaced by NP on line 13.

Following the AS assignment, the bulk of built-in optimization passes (e.g., canonicalization and loop optimizations) are executed on the resulting IR.

4.3.3 Legalizing accesses to host pointers

Before the late built-in optimizations, our second pass utilizes the AS assignments to legalize pointers before the IR is passed on to the back-end. *Legalization* is the process of converting generic IR types and operations to target-specific ones supported by the back-end. As all major ISAs today treat pointers like integers in terms of operations and register storage, pointers are just fixed-width integers in the back-end. The LLVM back-end can legalize operations on wider-than-native *integers* to multiple native instructions on multiple native registers. However, it can not generically legalize memory accesses to wider-than-native addresses.

Algorithm 4 Legalizing accesses to host pointers.

```
1:  for each module M do
2:      for each load L ∈ M do
3:          if L loads from host AS then
4:              SZ ← size in bits of L
5:              IA ← integer address of L
6:              replace L with wide_load call of SZ to IA
7:          end if
8:      end for
9:      for each store S ∈ M do
10:         if S stores into host AS then
11:             SZ ← size in bits of S
12:             IA ← integer address of S
13:             replace S with wide_store call of SZ to IA
14:         end if
15:     end for
16:     for each addrspacecast A ∈ M do
17:         SA ← integer source address of A
18:         if SA in device AS then
19:             DA ← zero extension of SA
20:         else if SA in host AS then
21:             DA ← truncation of SA
22:         end if
23:         replace A with pointer from address DA
24:     end for
25: end for
```

This problem is solved by our `HostPointerLegalizer` pass, shown in Algorithm 4, by utilizing the assigned address spaces. Wide host pointer accesses are legalized by replacing them with calls to builtin functions `wide_load()` and `wide_store()` (as introduced in § 3, and to be defined in § 5), by replacing the wide pointers with fixed-width *integer* types that the back-end can already legalize. Specifically, the pass does the following for every module in device code. On lines 6 and 13, it replaces all loads from the host AS with calls to the `wide_load` function and all stores to the host AS with `wide_store` calls. Note that memory can additionally be accessed through intrinsic functions (e.g., *memcpy*), and the device RTL needs to provide implementations of these functions that can work with host-AS arguments. Finally, on line 23, the pass resolves AS casts from device to host AS by zero extension and from host to device AS by truncation of the source address. This truncation is lossless as Algorithm 2 ensured that the pointer only ever holds inner pointers in the device AS.

Once pointers are legalized, late built-in optimizations (e.g., target specialization) can be applied before the IR is passed on to the target-specific back-end. Importantly, the *inline* pass is executed at this stage, to minimize the performance impact of the `wide_load()` and `wide_store()` functions.

Option	Requires mod. ISA	Requires mod. Core	Instrs.	Cycles
Wider loads & stores	Y	Y	1	$L + 1$
Adding CSRs	N	Y	4	$L + L' + 2$
Mem.-mapped ext. reg.	N	N	6	$L + L' + 4$

Table 2. Alternatives for accessing memory addresses wider than the data width of a core. The two right-most columns quantify the instructions and number of cycles of each alternative for loading (storing) a 64-bit value from (to) a 64-bit address with a 32-bit core. L is the latency of the first (or only) memory access, L' the latency of the subsequent access with an offset of 4 on the same base address. Modifications to the ISA also require modifications to the compiler backend.

5 HW Support for Extended Addressing

In the previous sections we left the `wide_load()` and `wide_store()` functions, which implement address-extended loads and stores, as black boxes. We will now define them. There are several options to implement this functionality in the underlying hardware. In this section, we present three options with decreasing degrees of intrusiveness, listed in Table 2, and implement the least intrusive option to show the generality of our solution and to upper-bound its overhead. For generality and because 64-bit addresses also induce a considerable amount of 64-bit *data* accesses, the examples discuss loading and storing 64-bit values. The reduction of the examples to 32-bit (and smaller) values with 64-bit addresses is trivial.

5.1 Additional, Wider Load & Store Instructions

The most intrusive option is to extend the ISA with custom load and store instructions that operate on paired registers. For example, a 32-bit ISA could be extended with instructions such as `ldd x0, 0(x2)` to load from `x3` (upper 32 address bits) and `x2` (lower 32 address bits) into the registers `x1` (upper 32 data bits) and `x0` (lower 32 address bits). Assuming a standard register file (RF) with two read and one write ports, each such load and store would take one extra cycle on top of the latency of the memory access, because the wider load needs to write the upper half of data to the RF and the wider store needs to read the upper half of address and data from the RF. Thus, this ISA extension allows a 32-bit core to access 64-bit addresses with one instruction and $L + 1$ cycles, where L is the latency of the access. The compiler backend would need to be modified to know the double-register semantics of such instructions. This option requires logic to decode the additional instructions and a state register to control the address extension within the core but no additional register to hold the address extension.

5.2 Additional Control and Status Registers (CSRs)

As extending the ISA might not be possible, a less intrusive option is to add control and status registers (CSRs) to hold the

part of an address that does not fit into registers. For example, one 32-bit CSR, which the LSU uses as upper 32 address bits, allows a 32-bit core to access 64-bit addresses. If the CSR is defined to clear on the next memory access and disable interrupts until the memory access (to prevent the address extension from corrupting memory accesses in interrupts), a 32-bit core can load a 64-bit value from a 64-bit address with the following four standard RISC-V[1] instructions:

```
csrrw  x0, csr_addr_ext, x3 // set upper half of address
                            // and disable interrupts
lw     x0, 0(x2)            // load lower half of data
                            // and reenable interrupts
csrrw  x0, csr_addr_ext, x3 // set upper half of address
                            // and disable interrupts
lw     x1, 4(x2)            // load upper half of data
                            // and reenable interrupts
```

where pre- and post-conditions on the registers are as in the last paragraph. The instructions setting the address extension CSR take one cycle each. The first `lw` might miss in the cache (latency L), while the second `lw` with an offset of four bytes to the same base address almost certainly hits (latency L'). Thus, this solution allows a 32-bit core with one additional 32-bit CSR to access a 64-bit address with 4 standard instructions and $L + L' + 2$ cycles.

5.3 Memory-Mapped External Register

The least intrusive option is to place an address extension register right outside the core and map it to the I/O address space of the core. For example, one 32-bit external register that extends the 32-bit address provided by the LSU of that core allows to access 64-bit addresses from an unmodified 32-bit core. Like the CSR, this register is defined to clear on the next memory access. A load with the same semantics as in the other examples can be performed with the following six standard RISC-V instructions:

```
csrrci x4, csr_status, 3    // disable interrupts
sw     x3, 0(mem_addr_ext)  // set upper (sic!) half of address
lw     x0, 0(x2)            // load lower half of data
sw     x3, 0(mem_addr_ext)  // set upper half of address
lw     x1, 4(x2)            // load upper half of data
csrrw  x0, csr_status, x4   // reenable interrupts
```

Like a CSR, a register directly after the core can generally be accessed in one cycle. Thus, this solution allows an entirely unmodified 32-bit core to access a 64-bit address with 6 standard instructions and $L + L' + 4$ cycles. The required extra hardware is one 32-bit register outside the core.

We implement this last option as it is the most generic and puts an upper bound on the overhead of our solution.

[1]Similar constructs are possible in other ISAs, we use RISC-V as a concrete example.

6 Evaluation

We show that our solution enables OpenMP offloading across data model boundaries with an average run-time overhead below 0.7 % compared to offloading restricted, native-accelerator-width addresses over a wide range of benchmarks.

6.1 Methodology

We implement our compiler in LLVM 8.0.0 [30], and we use a custom version of the open-source HERO heterogeneous research platform [27, 28] to implement extended addressing as described in § 5.3. We use a 64-bit RISC-V Ariane core [63] as host and a cluster from the PULP project [49] with 8 32-bit RISC-V PEs [18], one DMA engine, and 256 KiB of L1 SPM in 16 banks that the PEs can access in a single cycle, as PMCA. Each PE has a memory-mapped external 32-bit register to extend addresses to 64 bits. All hardware is implemented in synthesizable hardware description language (HDL) and benchmarks are measured in cycle-accurate hardware simulation using Questa 10.7b [35].

Kernel	Parallelized computation	Complexity space	Complexity comput.
`2mm`	$C_{i,j} = \sum_{k=1}^{N} \alpha A_{i,k} B_{k,j}$	$O(N^2)$	$O(N^3)$
`3mm`	$E = \texttt{2mm}(A, B) \rightarrow F = \texttt{2mm}(C, D) \rightarrow G = \texttt{2mm}(E, F)$	$O(N^2)$	$O(N^3)$
`atax`	$B_i = \sum_{j=1}^{N} A_{i,j} X_j \rightarrow Y_i = \sum_{j=1}^{N} A_{j,i} B_j$	$O(N^2)$	$O(N^2)$
`bicg`	$Q_i = \sum_{j=1}^{N} A_{i,j} P_j \rightarrow S_j = \sum_{i=1}^{N} R_i A_{i,j}$	$O(N^2)$	$O(N^2)$
`conv2d`	$B_{i,j} = \sum_{(k,l)=(-1,-1)}^{(1,1)} c_{k,l} A_{i+k,j+l}$	$O(N^2)$	$O(N^2)$
`covar`	$E_j = \alpha \sum_{i=1}^{M} D_{i,j}; D_{i,j} \mathrel{-}= E_j; S_{i,j} = S_{j,i} = \sum_{k=1}^{N} D_{k,i} D_{k,j}$	$O(N^2)$	$O(N^3)$
`gemm`	$C_{i,j} = \beta \left(\sum_{k=1}^{N} \alpha A_{i,k} B_{k,j} \right)$	$O(N^2)$	$O(N^3)$

Table 3. Evaluated kernels. Subscripts denote indices, uppercase letters are variables, and lowercase letters are constants. Arrows (→) denote consecutive offloads. Semicolons (;) denote consecutive parallel phases within the same offload.

We evaluate the seven kernels listed in Table 3. From the Polybench/ACC benchmark suite [19], `2mm`, `3mm`, `atax`, `bicg`, and `gemm` are linear algebra kernels, `conv2d` is part of the "stencil" domain, and `covar` is part of the "datamining" domain. Together, these commonly accelerated kernels span a wide range of memory access patterns and operational intensities. All matrices are stored in row-major arrays. Data is copied to and from accelerator L1 SPM with the DMA engine at the beginning and end of each offload phase, respectively. Computations of the accelerator thus exclusively use the L1 SPM. The accelerator PEs execute the computation in the second column of Table 3 in parallel. `3mm`, `atax`,

and `bicg` are composed of consecutive offloads, denoted by arrows (→) in the table, all other kernels consist of a single offload. All benchmarks are compiled with `-O3` but no specific optimization flags.

We measure the run time of each kernel in accelerator clock cycles, starting before the first DMA transfer of input data and stopping after the last DMA transfer of output data. In the *baseline*, the accelerator works exclusively with 32-bit addresses, i.e., the benchmarks are compiled for a 32-bit host and accelerator. We compare two implementations, where the benchmarks are compiled for a 64-bit host and a 32-bit accelerator with 64-bit generic AS, to the baseline: First, to analyze the performance impact of handling 64-bit pointers within the kernels on the 32-bit accelerator, we do not run the AS assignment pass. In other words, all accesses require 64-bit extension and take $L + L' + 4$ cycles, as described in § 5.3. Second, to analyze the efficiency of our solution, we run our full compiler including the AS assignment pass.

6.2 Benchmark Results

As the performance-critical part of each benchmark operates exclusively on device memory, which can be addressed with 32-bit pointers, our hypothesis is that the run-time overhead of our mixed-data-model compiler converges to zero with increasing data sizes. The evaluation is focused on small data sizes to analyze the effect on fine-grained offloading and the rate of convergence.

Fig. 5 shows the execution time of all benchmarks and both implementations relative to the baseline, where the accelerator works exclusively with native 32-bit addresses. For each benchmark, four bars represent different data sizes; for example, size 8 means that all matrices in a benchmark are 8×8 and vectors have length 8.

In the left part, where the accelerator has to work with 64-bit addresses also for local memory, the run time is multiplied by a factor of 1.4 to 5.8. For `3mm`, `2mm`, `gemm`, and `covar`, the relative run time converges to more than 4×. Those three kernels are dominated by computations and thus also by local memory accesses by the PEs using `wide_load/store`, and each access to a 32-bit word in L1 now takes 4 instead of 1 cycle. In addition, the `wide_*` memory accesses leave the compiler less freedom for scheduling memory accesses: it is currently not possible to define ordering constraints related to 64-bit addresses in a 32-bit compiler, so the order of every `wide_*` with respect to *any* other memory access needs to be preserved. The overhead is less pronounced for the other kernels, which are more balanced between data transfers through the DMA engine and memory accesses by the PEs. Nonetheless, the run-time overhead of using a 64-bit AS as generic AS for the device is clearly prohibitive, constituting the need for our AS assignment compiler pass.

In the right part of Fig. 5, where our compiler pass assigns as many device pointers to the 32-bit device AS as possible, the situation is completely different. Even for very small data sets (8×8 matrices and 8×1 vectors), the run-time overhead never exceeds 22 %. Even more importantly, the overhead rapidly converges to zero for all kernels, and already is below 0.7 % on average for still small data sets of size 64. This demonstrates the effectiveness of our AS assignment pass and proves that our compiler enables mixed-data-model offloading with negligible overheads in run time.

7 Related Work

To our knowledge, no computer today supports sharing data in the full host address space with accelerators that have a shorter data width. The concept of passing pointers as fixed-width integers (see § 3) could apply to such computers as well, given minimal hardware support for extended addressing (see § 5). However, we know no related work that includes these contributions (and consequently neither the compiler in § 4). In this section, we discuss how related works implement offloading by avoiding mixed data models.

Existing heterogeneous computers with mixed-width components (e.g., [9, 17, 23, 28, 48]) do not support OpenMP offloading or restrict the address space of offloaded applications to that of the accelerator (see § 2.4).

Most general-purpose GPUs (GPGPUs) used today in heterogeneous computers can natively access 64-bit addresses [1, 41]. GPGPUs, which implement double-precision floating-point arithmetics in hardware and are designed for SIMD (and SIMT) parallelism, naturally have a wide data path, so 64-bit addressing comes at very little additional cost. This also applies to CPU+GPU SoCs; for example, Nvidia's Tegra today fully supports offloading in 64-bit applications [40], although earlier versions restricted the address space of offloaded applications to 32 bit [39]. Many works address OpenMP offloading from 64-bit hosts to 64-bit GPUs with LLVM [2, 4, 45].

Digital signal processors (DSPs) are an important class of accelerators that benefit much less from a 64-bit data path because they usually operate on sensor data, which does not exceed 32 bit in precision (although DSPs are also designed for SIMD parallelism). Today, even high-end DSPs are 32-bit machines [54] and the SoCs they are used in feature 32-bit host processors [52], even for driver assistance systems that include graphics accelerators [53]. Many of these fully-32-bit DSP SoCs support OpenMP offloading [37, 51]. Other DSPs [5, 10, 17] and DSP-like accelerators [13, 43] that feature 32-bit very long instruction word (VLIW) ISAs are typically programmed through application-specific libraries such as OpenCV [46] and OpenVX [26]. Provided a minimal OpenMP device RTL and support for the ISA in LLVM, our work could enable to use these accelerators as OpenMP offload targets in modern SoCs with 64-bit host processors.

Address spaces are also used in CUDA [11] to define the memory location of functions and variables. Clang compiles CUDA code with address specifiers such as `__device__` to

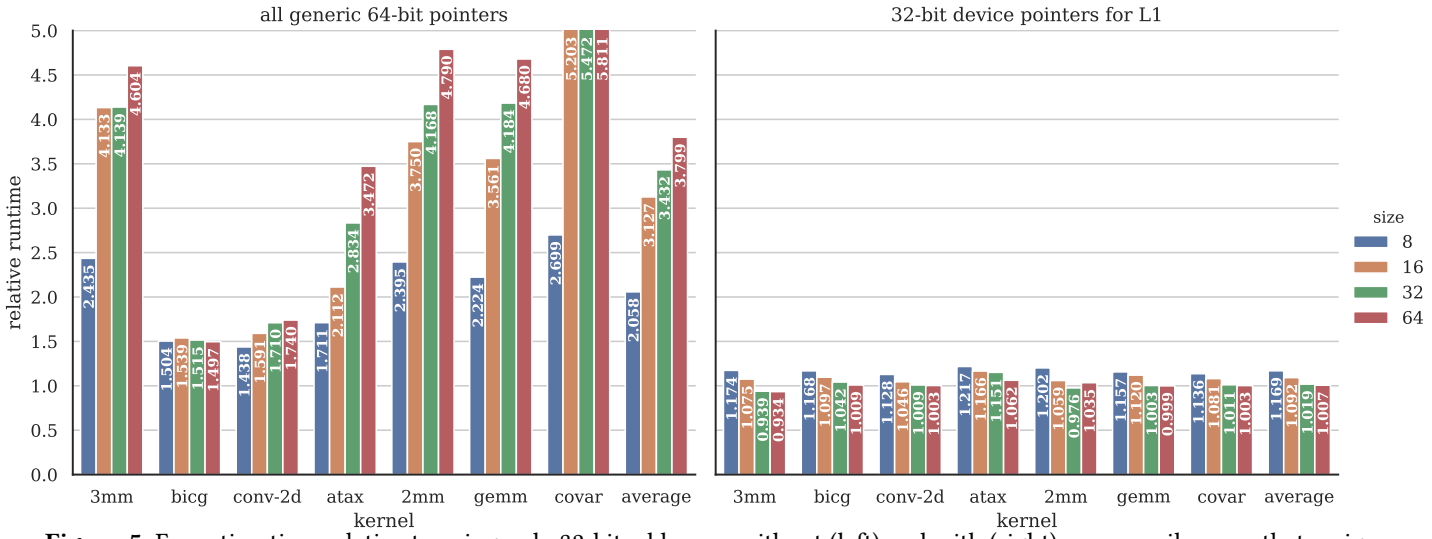

Figure 5. Execution time relative to using only 32-bit addresses without (left) and with (right) our compiler pass that assigns device pointers to the 32-bit device address space.

`addrspace` attributes in LLVM IR. In contrast to our work, no two CUDA address spaces overlap, so they are not used for accessing host memory from the device (or vice-versa) [42]. Address spaces in LLVM can further define pointers to be non-integral, which has been used to implement "fat" pointers with hardware support for memory protection [8, 62].

OpenACC [7] is a heterogeneous programming alternative to OpenMP. Its data model [61] is more generic than that of OpenMP but also supports accelerator-private memory. In Clang, OpenACC is being implemented by translating to OpenMP [15], so our work naturally extends to OpenACC as far as implemented in Clang.

Different approaches enable automatic offloading to GPUs. Graphite-OpenCL [47] first proposed static offloading of parallel loops to GPUs, relying on polyhedral analysis techniques to identify suitable subprograms. In the context of LLVM, several approaches use Polly [20] as a foundation for automatic accelerator mapping. Examples are KernelGen [36] which introduced a device focused approach only falling back to the host system if unavoidable, Damschen et al.'s approach [14] using a sophisticated client-server approach to orchestrate computations on Xeon Phi systems, and Polly-ACC [21] that introduces cross-kernel analysis to reduce overall data movement. To our understanding, all approaches target 64-bit devices and do not address offloading to devices with a data width that differs from that of the host. As our proposed concept for mixed-data-model compilation and offloading is not restricted to OpenMP but relies on generic IR analysis and transformations, it could apply to OpenCL and related frameworks as well.

Extended addressing has been implemented in processors for different purposes. In x86's Physical Address Extension (PAE) [50], page table entries are 64-bit but the (virtual) addresses used by processors remain 32 bit. Using address translation to access a wider host address space from accelerators is theoretically possible, but maintaining a virtual address space that is different from that of the host is not trivial, so we prefer simple address extension. Similar to the first of our address extension options, [57] extended a 32-bit ISA with 64-bit load and store instructions and added a special-purpose register for address extension (whereas our first option does not require additional registers). They observe that SPEC CPU2006 uses less than 4 GiB memory and thus use 32-bit load/store instructions whenever possible. The authors have recently integrated that work into a proposed composite ISA [56], which includes a 32-bit instruction subset. The focus of our work, in contrast, is to enable the first-class integration of 32-bit accelerators in a 64-bit addressed computer, and we design and implement compiler optimizations to do this in prevalent heterogeneous programming models and without restrictions or assumptions on the used memory space.

8 Conclusion

Our work extends prevalent programming models for heterogeneous computers (e.g., OpenMP, OpenACC) to computers with mixed data widths (e.g., 64-bit host and 32-bit accelerator) for the first time. We presented the general concept of mixed-data-model offloading in § 3 and designed and implemented an LLVM-based compiler to implement our solution fully transparently to the programmer in § 4. We discussed hardware support for extended addressing in § 5 and implemented the least intrusive variant to show the generality of our solution and upper-bound its overhead. Results on benchmarks from the PolyBench-ACC suite show that a 32-bit accelerator can transparently share memory with a 64-bit

host at an average overhead below 0.7 % compared to 32-bit-only execution, enabling mixed-data-model systems to execute at near-native performance.

References

[1] AMD Corp. 2018. AMD Radeon Instinct MI60. Datasheet. https://www.amd.com/system/files/documents/radeon-instinct-mi60-datasheet.pdf

[2] Samuel F. Antao, Alexey Bataev, Arpith C. Jacob, Gheorghe-Teodor Bercea, Alexandre E. Eichenberger, Georgios Rokos, Matt Martineau, Tian Jin, Guray Ozen, Zehra Sura, Tong Chen, Hyojin Sung, Carlo Bertolli, and Kevin O'Brien. 2016. Offloading Support for OpenMP in Clang and LLVM. In *LLVM-HPC'16*. https://doi.org/10.1109/LLVM-HPC.2016.6

[3] Arm Ltd. 2019. *Architecture Reference Manual: ARMv8 for ARMv8-A architecture profile*. Chapter D1.19 Interprocessing.

[4] Gheorghe-Teodor Bercea, Carlo Bertolli, Arpith C. Jacob, Alexandre Eichenberger, Alexey Bataev, Georgios Rokos, Hyojin Sung, Tong Chen, and Kevin O'Brien. 2017. Implementing Implicit OpenMP Data Sharing on GPUs. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC'17)*. ACM, New York, NY, USA, Article 5, 12 pages. https://doi.org/10.1145/3148173.3148189

[5] Cadence Design Systems, Inc. 2018. Tensilica Xtensa LX7 processor datasheet. https://ip.cadence.com/uploads/1099/TIP_PB_Xtensa_lx7_FINAL-pdf

[6] Alessandro Capotondi and Andrea Marongiu. 2017. Enabling Zero-copy OpenMP Offloading on the PULP Many-core Accelerator. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems (SCOPES '17)*. ACM, New York, NY, USA, 68–71. https://doi.org/10.1145/3078659.3079071

[7] S. Chandrasekaran and G. Juckeland. 2017. *OpenACC for Programmers: Concepts and Strategies*. Pearson Education.

[8] David Chisnall. 2015. Adventures with LLVM in a magical land where pointers are not integers. In *2015 LLVM Developer's Meeting*. https://llvm.org/devmtg/2015-02/slides/chisnall-pointers-not-int.pdf

[9] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 109, 6 pages. https://doi.org/10.1145/2897937.2897972

[10] Lucian Codrescu. 2015. Architecture of the Hexagon 680 DSP for Mobile Imaging and Computer Vision. In *2015 IEEE International Symposium on High Performance Chips (HOTCHIPS '27)*. https://www.hotchips.org/wp-content/uploads/hc_archives/hc27/HC27.24-Monday-Epub/HC27.24.20-Multimedia-Epub/HC27.24.211-Hexagon680-Codrescu-Qualcomm.pdf

[11] S. Cook. 2012. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*. Elsevier Science.

[12] E. G. Cota, P. Mantovani, G. Di Guglielmo, and L. P. Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744794

[13] Ian Cutress. 2016. CEVA Launches Fifth-Generation Machine Learning Image and Vision DSP Solution: CEVA-XM6. https://www.anandtech.com/show/10700

[14] Marvin Damschen, Heinrich Riebler, Gavin Vaz, and Christian Plessl. 2015. Transparent Offloading of Computational Hotspots from Binary Code to Xeon Phi. In *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exh (DATE '15)*. EDA Consortium, 1078–1083. http://dl.acm.org/citation.cfm?id=2757012.2757063

[15] Joel E Denny, Seyong Lee, and Jeffrey S Vetter. 2018. Clacc: Translating OpenACC to OpenMP in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 18–29.

[16] Michael Ditty, Ashish Karandikar, and David Reed. 2018. Nvidia's Xavier SoC. In *2018 IEEE International Symposium on High Performance Chips (HOTCHIPS '30)*. https://www.hotchips.org/hc30/1conf/1.12_Nvidia_XavierHotchips2018Final_814.pdf

[17] Andrei Frumusanu. 2018. The Qualcomm Snapdragon 855 Pre-Dive: Going Into Detail on 2019's Flagship Android SoC. https://www.anandtech.com/show/13680/snapdragon-855-going-into-detail

[18] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K Gürkaynak, and Luca Benini. 2017. Near-threshold RISC-V core with DSP extensions for scalable IoT endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 10 (2017), 2700–2713.

[19] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. 2012. Auto-tuning a high-level language targeted to GPU codes. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–10.

[20] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters* 22, 04 (2012), 1250010.

[21] Tobias Grosser and Torsten Hoefler. 2016. Polly-ACC Transparent Compilation to Heterogeneous Hardware. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*. ACM, New York, NY, USA, Article 1, 13 pages. https://doi.org/10.1145/2925426.2926286

[22] J.L. Hennessy and D.A. Patterson. 2017. *Computer Architecture: A Quantitative Approach*. Elsevier Science, Chapter 7.2 Guidelines for Domain-Specific Architectures.

[23] N. Jouppi, C. Young, N. Patil, and D. Patterson. 2018. Motivation for and Evaluation of the First Tensor Processing Unit. *IEEE Micro* 38, 3 (May 2018), 10–19. https://doi.org/10.1109/MM.2018.032271057

[24] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 1–12. https://doi.org/10.1145/3079856.3080246

[25] M. Kerrisk. 2010. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.

[26] Khronos Group Inc. 2019. OpenVX API Specification 1.3. https://www.khronos.org/registry/OpenVX/specs/1.3/OpenVX_Specification_1_3.pdf

[27] Andreas Kurth, Alessandro Capotondi, Pirmin Vogel, Luca Benini, and Andrea Marongiu. 2018. HERO: An Open-Source Research Platform for HW/SW Exploration of Heterogeneous Manycore Systems. In *Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems (ANDARE '18)*. ACM, New York, NY, USA, Article 5, 6 pages. https://doi.org/10.1145/3295816.3295821

[28] Andreas Kurth, Pirmin Vogel, Alessandro Capotondi, Andrea Marongiu, and Luca Benini. 2017. HERO: Heterogeneous Embedded Research Platform for Exploring RISC-V Manycore Accelerators on FPGA. In *Computer Architecture Research with RISC-V (CARRV '17)*.

[29] A. Kurth, P. Vogel, A. Marongiu, and L. Benini. 2018. Scalable and Efficient Virtual Memory Sharing in Heterogeneous SoCs with TLB Prefetching and MMU-Aware DMA Engine. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. 292–300. https://doi.org/10.1109/ICCD.2018.00052

[30] LLVM. 2019. *LLVM 8.0.0 Release Notes.* https://releases.llvm.org/8.0.0/docs/ReleaseNotes.html

[31] H. J. Lu, H. Peter Anvin, and Milind Girkar. 2011. X32: A native 32-bit ABI for x86-64. In *Linux Plumbers Conference.* http://www.linuxplumbersconf.net/2011/ocw/system/presentations/531/original/x32-LPC-2011-0906.pptx

[32] Andy Lutomirski. 2018. *Can we drop upstream Linux x32 support?* https://lkml.org/lkml/2018/12/10/1145

[33] Andrea Marongiu, Alessandro Capotondi, Giuseppe Tagliavini, and Luca Benini. 2015. Simplifying many-core-based heterogeneous SoC programming with offload directives. *IEEE Transactions on Industrial Informatics* 11, 4 (2015), 957–967.

[34] M. Martineau, S. McIntosh-Smith, and W. Gaudin. 2016. Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 338–347. https://doi.org/10.1109/IPDPSW.2016.70

[35] Mentor, a Siemens Business. 2019. *Questa Advanced Simulator.* https://www.mentor.com/products/fv/questa/

[36] Dmitry Mikushin, Nikolay Likhogrud, Eddy Z Zhang, and Christopher Bergström. 2014. KernelGen–The Design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops.* IEEE, 1011–1020.

[37] Gaurav Mitra, Eric Stotzer, Ajay Jayaraj, and Alistair P. Rendell. 2014. Implementation and Optimization of the OpenMP Accelerator Model for the TI Keystone II Architecture. In *Using and Improving OpenMP for Devices, Tasks, and More*, Luiz DeRose, Bronis R. de Supinski, Stephen L. Olivier, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer International Publishing, Cham, 202–214.

[38] Nvidia Corp. 2014. Summit and Sierra Supercomputers: An Inside Look at the U.S. Department of Energy's New Pre-Exascale Systems. http://www.teratec.eu/actu/calcul/Nvidia_Coral_White_Paper_Final_3_1.pdf

[39] Nvidia Corp. 2015. *Linux for Tegra R23.1.* https://developer.nvidia.com/embedded/linux-tegra-r231

[40] Nvidia Corp. 2016. *Linux for Tegra R24.1.* https://developer.nvidia.com/embedded/linux-tegra-r241

[41] Nvidia Corp. 2019. Nvidia TITAN RTX. Product Brief. https://www.nvidia.com/content/dam/en-zz/Solutions/titan/documents/titan-rtx-for-creators-us-nvidia-1011126-r6-web.pdf

[42] Nvidia Corp. 2019. *NVVM IR Specification 1.5.* https://docs.nvidia.com/cuda/nvvm-ir-spec/index.html

[43] Nate Oh. 2017. Intel Announces Movidius Myriad X VPU, Featuring 'Neural Compute Engine'. https://www.anandtech.com/show/11771/intel-announces-movidius-myriad-x-vpu

[44] OpenMP Architecture Review Board 2015. *OpenMP Application Programming Interface.* OpenMP Architecture Review Board. Version 4.5.

[45] G. Özen, S. Atzeni, M. Wolfe, A. Southwell, and G. Klimowicz. 2018. OpenMP GPU Offload in Flang and LLVM. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC).* 1–9. https://doi.org/10.1109/LLVM-HPC.2018.8639434

[46] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov, and Victor Eruhimov. 2012. Real-time computer vision with OpenCV. *Commun. ACM* 55, 6 (2012), 61–69. https://research.nvidia.com/sites/default/files/pubs/2012-06_Realtime-Computer-Vision/OpenCV_CACM_p61-pulli.pdf

[47] I RAS. 2010. GRAPHITE-OpenCL: Generate OpenCL code from parallel loops. *GCC Developers Summit. Citeseer* (2010), 9.

[48] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. 2018. Pixel Visual Core: Google's Fully Programmable Image, Vision, and AI Processor for Mobile Devices. In *2018 IEEE International Symposium on High Performance Chips (HOTCHIPS '30).* https://www.hotchips.org/hc30/1conf/1.02_Google_HC30.Google.JasonRedgrave.V01.pdf

[49] Davide Rossi, Igor Loi, Francesco Conti, Giuseppe Tagliavini, Antonio Pullini, and Andrea Marongiu. 2014. Energy efficient parallel computing on the PULP platform with support for OpenMP. In *Electrical & Electronics Engineers in Israel (IEEEI), 2014 IEEE 28th Convention of.* IEEE, 1–5.

[50] T. Shanley. 1998. *Pentium Pro and Pentium II System Architecture.* Addison-Wesley, Chapter 22 Paging Enhancements.

[51] Eric Stotzer, Ajay Jayaraj, Murtaza Ali, Arnon Friedmann, Gaurav Mitra, Alistair P. Rendell, and Ian Lintault. 2013. OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip. In *OpenMP in the Era of Low Power Devices and Accelerators*, Alistair P. Rendell, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–127.

[52] Texas Instruments Inc. 2019. 66AK2Hxx Multicore DSP+ARM KeyStone II System-on-Chip (SoC) datasheet (Rev. G). http://www.ti.com/lit/ds/symlink/66ak2h14.pdf

[53] Texas Instruments Inc. 2019. TDA2x ADAS Applications Processor 17mm Package (AAS) Silicon Revision 2.0 datasheet (Rev. F). http://www.ti.com/lit/ds/sprs952f/sprs952f.pdf

[54] Texas Instruments Inc. 2019. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor datasheet (Rev. E). http://www.ti.com/lit/ds/symlink/tms320c6678.pdf

[55] Sudharshan S. Vazhkudai, Bronis R. de Supinski, Arthur S. Bland, Al Geist, James Sexton, Jim Kahle, Christopher J. Zimmer, Scott Atchley, Sarp Oral, Don E. Maxwell, Veronica G. Vergara Larrea, Adam Bertsch, Robin Goldstone, Wayne Joubert, Chris Chambreau, David Appelhans, Robert Blackmore, Ben Casses, George Chochia, Gene Davison, Matthew A. Ezell, Tom Gooding, Elsa Gonsiorowski, Leopold Grinberg, Bill Hanson, Bill Hartner, Ian Karlin, Matthew L. Leininger, Dustin Leverman, Chris Marroquin, Adam Moody, Martin Ohmacht, Ramesh Pankajakshan, Fernando Pizzano, James H. Rogers, Bryan Rosenburg, Drew Schmidt, Mallikarjun Shankar, Feiyi Wang, Py Watson, Bob Walkup, Lance D. Weems, and Junqi Yin. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-exascale Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18).* IEEE Press, Piscataway, NJ, USA, Article 52, 12 pages. https://doi.org/10.1109/SC.2018.00055

[56] A. Venkat, H. Basavaraj, and D. M. Tullsen. 2019. Composite-ISA Cores: Enabling Multi-ISA Heterogeneity Using a Single ISA. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA).* 42–55. https://doi.org/10.1109/HPCA.2019.00026

[57] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14).* IEEE Press, Piscataway, NJ, USA, 121–132. http://dl.acm.org/citation.cfm?id=2665671.2665692

[58] Pirmin Vogel, Andreas Kurth, Johannes Weinbuch, Andrea Marongiu, and Luca Benini. 2017. Efficient virtual memory sharing via on-accelerator page table walking in heterogeneous embedded SoCs. *ACM Transactions on Embedded Computing Systems (TECS)* 16, 5s (2017), 154.

[59] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2018. Exploring shared virtual memory for FPGA accelerators with a configurable IOMMU. *IEEE Trans. Comput.* (2018).

[60] Andrew Waterman, Yunsup Lee, Rimas Avizienis, David A Patterson, and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.* Version 20190608-Priv-MSU-Ratified.

[61] M. Wolfe, S. Lee, J. Kim, X. Tian, R. Xu, S. Chandrasekaran, and B. Chapman. 2017. Implementing the OpenACC Data Model. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 662–672. https://doi.org/10.1109/IPDPSW.2017.85

[62] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 457–468. http://dl.acm.org/citation.cfm?id=2665671.2665740

[63] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-ready 1.7 GHz 64bit RISC-V Core in 22nm FDSOI Technology. *arXiv preprint arXiv:1904.05442* (2019).