RESEARCH-ARTICLE

Programmer productivity and performance on AMD's AI Engines: Offloading Fortran intrinsics via MLIR a case-study

NICK BROWN, The University of Edinburgh, Edinburgh, Scotland, U.K.

GABRIEL RODRIGUEZ-CANAL, The University of Edinburgh, Edinburgh, Scotland, U.K.

Programmer productivity and performance on AMD's AI Engines: Offloading Fortran intrinsics via MLIR a case-study

Nick Brown
EPCC
University of Edinburgh
Edinburgh, United Kingdom
n.brown@epcc.ed.ac.uk

Gabriel Rodriguez-Canal
EPCC
University of Edinburgh
Edinburgh, United Kingdom
gabriel.rodcanal@ed.ac.uk

Abstract

A major challenge the HPC community faces is how to deliver increased performance demanded by scientific programmers, whilst addressing an increased emphasis on sustainable operations. Specialised architectures, such as FPGAs and AMD's AI Engines (AIEs), have demonstrated significant energy efficiency advantages, however a major challenge is that substantial expertise and investment of time is required to gain best performance from this hardware which is a major blocker.

Fortran in the lingua franca of scientific computing, and in this paper we explore the automatic offloading of Fortran intrinsics to the AIEs in AMD's Ryzen AI CPU as a case study, demonstrating how the MLIR compiler ecosystem can provide performance and programmer productivity. We describe an approach that lowers the MLIR linear algebra dialect to AMD's AIE dialects, and demonstrate that for suitable workloads the AIEs can provide significant performance advantages over the CPU without any code modifications required by the programmer.

CCS Concepts

• **Computer systems organization** → **Reconfigurable computing**; **Data flow architectures**; • **Software and its engineering** → **Source code generation**; **Runtime environments**; **Compilers**.

Keywords

AMD AI engines, Versal Adaptive SoC, Ryzen AI, Fortran, MLIR, xDSL

1 Introduction

Whilst High Performance Computing, HPC, currently relies on mainstream CPUs and GPUs, driven by the incessant demand from scientific computing for increased performance, but at the same time an increased emphasis on energy efficiency, there is increasing interest in leveraging new more specialised hardware technologies. AMD Xilinx's AI engines are one example of this, initially released as part of the Versal Adaptive SoC these engines are vector arithmetic accelerators. AI Engines, or AIEs and we use these two terms interchangeably throughout this paper, adopt a Very Long Instruction Word (VLIW) design and contain a dedicated 512 bit vector unit. There are up to 400 engines on a Versal, and it has been demonstrated that this large amount of raw compute is suitable for a range of High Performance Computing (HPC) workloads [4] [18] [13].

In 2023 AMD released their Ryzen AI 7000 series of CPUs which was then followed in 2024 and 2025 by the AMD Ryzen AI 300 and AMD Ryzen AI Max series. These CPUs combine traditional x86 processing cores with a Neural Processing Unit (NPU) which is an array of second generation AIEs, known as AIE-ML, that are targeted more towards AI workloads. This very close coupling of the AIEs with the CPU cores offers a range of potential opportunities for leveraging these specialised engines to accelerate computational kernels, and furthermore the Ryzen AI CPU makes running on AIEs a much more attractive proposition for end users compared to buying a Versal.

For all these potential benefits, a major challenge with AIEs is in the programming of this technology. Developers must use a specialised API and possess a detailed knowledge of the architecture. Moreover, a significant time investment is required to port codes to the AIEs and optimise for performance. This is especially challenging given that kernels are written in C++ or Python, whereas over 60% of scientific computing workloads on a typical supercomputer are in Fortran [16].

It is our belief that, in order to drive adoption of AIEs, offloading from the CPU to the NPU must be entirely transparent to the programmer, requiring no effort on their behalf. Recent advances in compiler technologies has meant that folding much of this into the compiler is now a much more realistic proposition than ever before, and in this paper we describe an approach which enables the seamless offloading of specific Fortran intrinsic calls onto AMD's AI engines as a case study to explore this central idea. This paper is structured as follows; in Section 2 we explore the background to this work, before in Section 3 describing our approach to offloading Fortran intrinsics on the AIEs. This is then followed by a performance comparison against running these intrinsics on the CPU in Section 4, before drawing conclusions and discussing further work in Section 5.

The contributions of this paper are as follows:

- To the best our our knowledge, the first demonstration that scientific programmers can transparently offload parts of their existing codes onto AIEs without any modifications.
- Description of a compilation approach where support for new AIE kernels can be trivially integrated.
- A performance exploration of the AMD Ryzen AI NPU for common Fortran intrinsics, demonstrating the NPU provides benefits for specific types of intrinsic.

2 Background and related work

AMD embedded, formerly Xilinx, developed AI Engines to harden commonly used arithmetic operations and combined this with the flexibility of their reconfigurable fabric in the Versal. AIEs follow a Very Long Instruction Word (VLIW) approach where, each cycle, they are capable of issuing up to seven instructions, and an AIE can handle both scalar and vector operations, with the vector unit of size 512 bits. AIEs are arranged in a 2D array, with engines connected to their neighbours in both dimensions, and are able to access the memory from their north, south and west neighbours directly. Furthermore, each engine has four data movers comprising two 32 bit input streams and two 32 bit output streams.

A natural evolution of this technology was to integrate AIEs directly into the CPU, where the CPU contains a Neural Processing Unit (NPU) which is an array of 20 AIEs arranged in five columns of four rows. Each row also contains a memory tile, and four of the columns have an interface tile which connects it to the CPU. The AI engines in the Ryzen AI NPU are the AIE-ML series which, in contrast to the AIEs in the Versal, have been more heavily optimised for AI workloads. Consequently, the AIE-ML contains double the data memory per AIE, 64KB and, unlike the Versal's AIE array, there are five dedicated memory tiles, one per column. Each of these contains 512KB of memory which is decomposed across 16 banks. Each memory tile is equipped with 12 data movers providing a total bandwidth of up to 30 GB/s [2]. This enhancement to the memory contained within the AIE array provides additional flexibility and a wider set of workloads that can be supported.

AMD have however also removed some arithmetic support in the AIE-ML compared to the first generation AIE, for example the vector units in the AIE-ML series do not natively support int32 or float32 data types in hardware, although these can be emulated. Instead, bfloat16 is provided and int32 integer arithmetic is only natively supported when mixed with int16. Whilst this is a limitation for HPC workloads, the AI engine technology is evolving rapidly and coupling of x86 CPU cores with the NPU is very promising. Consequently, it is still interesting to explore this technology in the context of HPC as future versions could provide increased precision if there is a market demand.

The code that runs on the AIEs comprises of two parts, kernels which are mapped to the AI engines themselves and a graph description which connects kernels and memories together via streams. Kernels operate following the consumer-producer model, where they consume input data from a maximum of two streams and produce results on a maximum of two output streams. These streams can be connected to the CPU directly via an interface tile, to a memory tile or to another AI engine compute tile.

There have been some efforts to address programmer productivity on the AIEs, for instance in [15] the authors presented an end to end programming model for AIEs and a Python embedded Domain Specific Language (DSL). In [12] the HPX programming framework was enhanced to support AI engines by extending the TaPaSCo framework enabling TaPaSCo FPGA and AIE tasks to be transparently integrated into HPX applications. TaPaSCo [9] is an open source toolchain that provides a scriptable flow for the construction of dataflow designs on FPGAs, and an APIs that provides task parallel computing on FPGAs. This tool was enhanced [10] to also support programming AIEs on the Versal and integrates with the existing FPGA approach. However, all these tools share the same limitation that programmers must learn new technologies and then manually port their codes, potentially requiring rewriting in new languages. By contrast, our opinion is that new hardware, such as the AIEs, must meet programmers where they are already at, in short without requiring any changes to existing codes or specialised knowledge on behalf of the programmer.

2.1 LLVM and MLIR

LLVM [14] provides reusable compiler and tool chain technologies the enable the development of compilers across different languages and hardware. There are many language frontends provided by LLVM and support for a range of hardware backends, with these connected via LLVM-IR. Consequently, an LLVM frontend such as Clang, which provides C and C++, that generates LLVM-IR is able to target any backend, and support for a wide range of architectures including CPUs, GPUs, and FPGAs has been developed. However, LLVM-IR is low level, and it requires significant work by each frontend to target LLVM-IR and results in duplication of compilation infrastructure between frontends.

MLIR, which was first developed by Google and then released open source in 2019, aims to address this issue of duplication by providing many IR dialects and transformations between them. Consequently, instead of targeting the low level LLVM-IR, frontends can translate to a mix of higher level intermediate representations and then leverage existing transformations within MLIR to lower to LLVM-IR. The IR follows a Static Single Assignment (SSA) form, and one of the major strengths of MLIR is that dialects can be mixed and manipulated separately, enabling progressive lowering of the abstraction level ultimately to LLVM-IR. Because this involves existing dialects and transformations, the MLIR approach enables a much greater sharing of compiler infrastructure between frontends, significantly reducing the overall software effort in developing compilers. MLIR also provides a framework for defining bespoke dialects and transformations.

MLIR is a sub-project of LLVM, and there are many IR dialects provided as standard including *memref* for memory management and data access, *func* to represent functions and calling between them, and *linalg* which express linear algebra operations. All of these ultimately lower to the *llvm* MLIR dialect, from which LLVM-IR is generated by the *mlir-translate* tool. A considerable community has grown up around MLIR with involvement from many vendors. AMD have invested heavily in this technology with their own fork of MLIR which targets the AIEs. AMD developed several dialects,

including *aie* that for streaming connections between AIE compute tiles and direct memory access, *aievec* for vector arithmetic operations, and *adf* to express AMD's Adaptive Data Flow (ADF) graph that connects tiles. For each of these, transformations and optimisations have been developed which ultimately results in a set of instructions that execute across the AIE array.

2.1.1 xDSL. One of the disadvantages of MLIR is that programmers must initially learn a range of complex LLVM concepts, and then work with the Tablegen format to describe dialects and keep track of the fast evolving MLIR repository. By contrast, xDSL [17] is a Python based compiler design toolkit which is 1-1 compatible with MLIR. Providing the majority of core MLIR dialects, as well as numerous additional experimental ones too, these are all expressed in the IRDL [6] format within Python classes. xDSL enables rapid exploration and prototyping of MLIR concepts, and once these are matured and proven they can then be contributed into the main MLIR codebase more easily, with the MPI dialect [3] being an example of this. Because xDSL is 1-1 compatible with MLIR, one is able to arbitrarily go between these technologies during compilation. We have used xDSL to develop the work described in this paper.

2.2 Flang

Flang [8] is LLVM's Fortran frontend and built on-top of MLIR. It began in 2020 as a ground-up rewrite of the previous Flang Fortran compiler, classic Flang, and is now an official component of LLVM. Whilst the objective of Flang is to support the full range of standard Fortran, including being able to adapt to future versions of the language, at the time of writing support for Fortran at or beyond 2003 is still work in progress although Flang is developing rapidly.
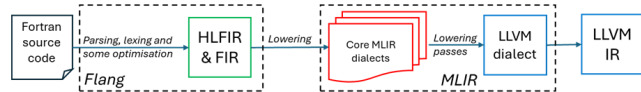

Figure 1: Illustration of MLIR-based Fortran compilation flow developed by [5] and based upon Flang to generate LLVM-IR.

Figure 1 illustrates a sketch of the overarching Flang compilation flow from [5] where, after lexing and parsing of a user's Fortran code, some optimisations are undertaken on the AST which is then lowered to Flang's High Level Fortran IR (HLFIR) [11] and Fortran IR (FIR) [7] dialects. These two MLIR dialects are part of Flang and specifically represent Fortran constructs and concepts. However, Flang is isolated from much of the rest of the MLIR ecosystem and the compiler then directly generates LLVM-IR from these two dialects. In this paper we leverage the work undertaken by [5] where, instead of directly generating LLVM-IR from these two dialects, a transformation pass lowers to the core MLIR dialects and then relies on the rest of the MLIR ecosystem to progressively lower and optimise the IR to generate the LLVM-IR. Not only does this approach enable integration with a wide range of existing infrastructure which is developed by the large MLIR community, moreover the IR can be intercepted at any point and specialised for specific architectures, in this paper for the AMD AIE.

In this paper we focus on Fortran intrinsics, which are built in procedures defined by the Fortran standard to provide utility functionality. Given Fortran's lineage in scientific computing, a range of intrinsics are defined that undertake calculations and examples include the *sum* procedure which sums all numbers in an array. Whilst the Flang compiler maps these directly to function calls in the Flang runtime library, the work undertaken in [5] instead mapped these to the linear algebra, *linalg*, dialect. Operations in this *linalg* dialect are then lowered using the existing MLIR infrastructure, for instance to be optimised for the CPU, and the rich source of information about the linear algebra operation can be leveraged to target other architectures, in this work AMD's AIEs.

3 Offloading Fortran intrinsics to AIEs

To offload Fortran intrinsics to the AIEs we have developed transformations and lowerings for operations in the linear algebra, *linalg*, dialect that target the Neural Processing Unit (NPU) in the Ryzen AI 7000 CPU. Figure 2 illustrates our approach, where the initial steps of transforming HLFIR and FIR dialects to the core MLIR dialects, including *linalg* for Fortran intrinsics, leverages the flow in Figure 1.

We developed a transformation that analyses operations in the *linalg* dialect and categorises these according to the functionality being undertaken. This is trivial where there is a direct mapping from the intrinsic to the *linalg* dialect's operation, for example matrix multiplication is represented as `linalg.matmul`. However other intrinsics are represented indirectly, for example the *sum* intrinsic results in the `linalg.reduce` operation, and this IR is illustrated in Listing 1 for accumulating over a one dimensional array. The body of the operation in Listing 1 operate across each element, with an individual element held in %32 and the running total in %33. Consequently, where appropriate, our transformation interrogates the body of linear algebra operations, such as a reduction, and categorises it accordingly.

```
linalg.reduce ins(%29:memref<?xi32>)
              outs(%30:memref<i32>) dimensions = [0]
  (%32 : i32, %33 : i32) {
    %34 = arith.addi %32, %33 : i32
    linalg.yield %34 : i32
}
```

Listing 1: Sketch of the IR corresponding to the Fortran sum intrinsic which leverages the `linalg.reduce` operation.

Once the specific type of linear algebra operation has been identified then, as illustrated in Figure 2, our flow generates two components; the host CPU side IR that both drives the NPU and contains the rest of the Fortran code, and the AIE side IR that is the *linalg* operation to run on the AIEs in the NPU. The CPU uses AMD's Xilinx RunTime (XRT) to manage and interact with the AIE array, and to aid this we developed an XRT wrapper MLIR dialect, *xrtw*, because there is no existing MLIR dialect that drives the AIEs from the CPU. In addition to the dialect, we also developed a transformation that lowers this to function calls, calling into the corresponding runtime library functions on the CPU via MLIR's `func.call` operation.
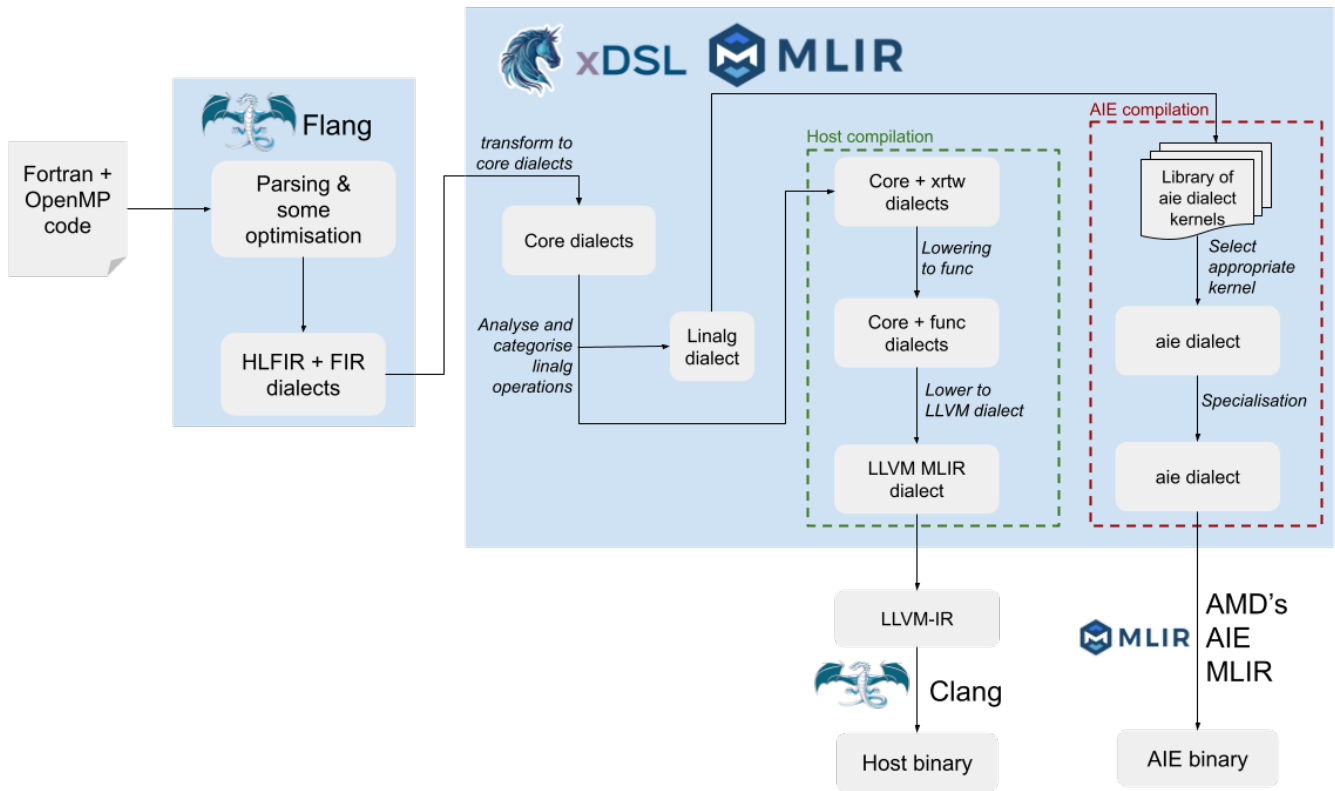
Figure 2: Illustration of our overarching compiler approach for offloading selected Fortran intrinsics to AMD's AIE array

However integration with XRT was more complex than initially assumed due to name mangling of C++ function and object names, making it difficult to directly call these from the IR. Whilst XRT provides a C interface, this is incomplete. For instance, the C interface omits the *register_xclbin* function which is required to set up the AIE array. However, the C++ and C APIs are incompatible and-so as this registration function must be called on the device object which is in C++, this then requires the rest of the C++ API to be used. Consequently, we developed a runtime wrapper which is precompiled and linked against on the CPU. This provides a simplified C style interface to XRT that is straightforward to call from the IR, and internally leverages the XRT C++ functions.

It is this runtime wrapper that our XRT wrapper, *xrtw*, MLIR dialect is based upon, and the IR is lowered to calling functions of, with the main operations in this dialect as follows:

(1) `xrtw.`**`num_devices`** returns the number of NPUs present and is used at runtime to determine whether to launch the instrinsic on the AIE array or instead, if none is present, to run on the CPU.
(2) `xrtw.`**`allocate_buffer`** which allocates buffers on the NPU with an associated integer identifier which is used as a reference to the buffer in other operations.
(3) `xrtw.`**`buffer_map`** will return a memref representing host-side memory associated with a buffer.

(4) `xrtw.`**`buffer_sync`** synchronises between the host and device, the direction (host to device or device to host) is provided as a parameter argument.
(5) `xrtw.`**`run`** accepts any number of buffers which are passed to the AIE kernel and the kernel is then launched.
(6) `xrtw.`**`wait`** blocks for completion of an AIE kernel.

Whilst this only covers a subset of XRT functionality, it is sufficient for driving the NPU in this work and can be extended in the future if required. Once the *xrtw* dialect is lowered to function calls, this IR is then provided to core MLIR transformations, ultimately being lowered to the LLVM-IR MLIR dialect, which is provided to the *mlir-translate* tool that then generates LLVM-IR which is compiled into an object file by Clang.

The second component is the IR generated from the *linalg* operation that will run on the NPU and undertakes the actual computation. We ported AMD's AI engine MLIR dialects into xDSL to enable us to represent IR in these dialects using that tool. The approach taken is to build a library of IR, based upon common linear algebra operations, and the appropriate IR can then be selected and loaded during compilation by the compiler and then specialised for the specific problem size and data type. We followed this approach because, for a specific linear algebra operation, the IR is very similar and only requires small modifications for each instantiation. Consequently, we pre-generate templates of MLIR IR which corresponds to each linear algebra operation, load this into xDSL's IRDL format and then store this externally in a library

by serialising the IR using pickle. Whilst it is possible to manually write the MLIR IRs for each linear algebra operation, AMD provides a Python interface to their MLIR dialects and a tool that generates MLIR from this, along with a range of numerical examples [1].

Consequently we store these existing examples, and our own where required, within this library. The appropriate IR is then loaded during compilation via pickle and deserialised. This IR can be considered a template, and the *specialisation* transformation in Figure 2 then manipulates this IR to specialise it for each instantiation, for example, by replacing placeholders in the IR with the type of data and number of elements that are being computed with. The resulting IR is illustrated in Listing 2, for a scalar addition running on a single AIE compute tile which adds one to each input value. FIFOs are first created using the `aie.objectfifo` operation to link the interface and compute tiles, and then the `aie.core` operation directs that the code that will run on an individual AIE compute tile. The FIFOs are configured to be of size 32, int32, elements, and the computation running on the AIEs works in batches of 32 elements. In each batch the input and output FIFOs are acquired via `aie.objectfifo.acquire` and then within the `scf.for` loop the input value is loaded from the FIFO, used in the add calculation, and stored into the output FIFO. These FIFOs are then released after the loop has completed via `aie.objectfifo.release`. For brevity, the IR which issues DMA memory copies between the host and the interface tile has been omitted.

In the example of Listing 2 the entirety of a core's computation is held directly in the IR, whereas it is also possible to call into external functions via the `func.call` operation. This is useful for more complicated kernels where one can, for example, develop a C++ kernel and compile it into an object file via the chess compiler. Different versions of these external functions, for instance handling distinct data types, can be called and this is materialised during the specialisation step. In this manner, our compilation flow is able to leverage a wide range of existing AIE kernels, with it possible to develop new implementations and optimisations which can then be integrated with our compiler via the dialect library.

This IR is then fed into AMD's AIE MLIR tooling, which generates the resulting *xclbin* and instruction files that are then launched on the NPU by the CPU code. Our compilation flow also maintains the CPU implementation of the linear algebra operation, for instance leveraging this instead if the data type or data size is not appropriate to run on the NPU.

4 Results and evaluation

In the experiments reported throughout this section we run on a Ryzen AI 7940HS CPU equipped with 32GB of DRAM. This CPU contains 20 AIE-MLs, arranged in five columns each of four rows. Each column also comprises a memory tile and four of the columns contain an interface tile connecting to the CPU. When running across the AIE array we only leverage the four columns that contain an interface tile, hence we run across 16 AIEs unless otherwise stated. We use GCC version 13.2, Vitis version 2023.1, XRT version 2.18.0, Flang version 20.0.0 (based upon LLVM release 18.1.8), and the latest AIE-MLIR at the time of writing. All results are averaged over ten runs and AIE execution times include the overhead of transferring data to and from the NPU. All CPU code is compiled

```
aie.device(npu1_1col) {
    %i_tile = aie.tile(0, 0)
    %comp_tile = aie.tile(0, 2)
    aie.objectfifo @in0(%i_tile, {%comp_tile},
            2 : i32) : !aie.objectfifo<memref<32xi32>>
    aie.objectfifo @out0(%comp_tile, {%i_tile},
            2 : i32) : !aie.objectfifo<memref<32xi32>>
    %comp_core = aie.core(%comp_tile) {
      %c0 = arith.constant 0 : index
      %problem_size = arith.constant 32 : index
      scf.for %arg0 = %c0 to %problem_size {
        %0 = aie.objectfifo.acquire @in0(Consume, 1) :
                !aie.objectfifosubview<memref<32xi32>>
        %1 = aie.objectfifo.subview.access %0[0] :
                !aie.objectfifosubview<memref<32xi32>>
                -> memref<32xi32>
        %2 = aie.objectfifo.acquire @out0(Produce, 1) :
                !aie.objectfifosubview<memref<32xi32>>
        %3 = aie.objectfifo.subview.access %2[0] :
                !aie.objectfifosubview<memref<32xi32>>
                -> memref<32xi32>
        %c0_0 = arith.constant 0 : index
        %c32 = arith.constant 32 : index
        scf.for %arg1 = %c0_0 to %c32 {
          %4 = memref.load %1[%arg1] : memref<32xi32>
          %c1_i32 = arith.constant 1 : i32
          %5 = arith.addi %4, %c1_i32 : i32
          memref.store %5, %3[%arg1] : memref<32xi32>
        }
        aie.objectfifo.release @in0(Consume, 1)
        aie.objectfifo.release @out0(Produce, 1)
      }
      aie.end
    }
    ...
}
```

Listing 2: Sketch of IR using the AIE dialects that will run on the AIE array for a scalar add example.

at optimisation level three, and CPU comparison executables are from Flang using the linear algebra based CPU lowering in [5].

```
1  integer :: data(100000), result, i
2  do i=1, 100000
3      data(i)=i
4  end do
5  result=sum(data)
```

Listing 1: Example use of the Fortran sum intrinsic

We undertook a performance comparison for reduction based Fortran intrinsics, such as *sum* which accumulates the values held in an array. Listing 1 sketches the programmer's Fortran code for the *sum* intrinsic, where the *data* array is defined and initialised, and then the intrinsic accumulates its values and returns this in

Intrinsic	CPU runtime (us)				NPU runtime (us)				
	int16	int32	bfloat16	float32	int16	int32	bfloat16	float32	conv-bfloat16
sum	606	296	5187	962	3107	3207	3534	3511	8623
product	627	305	4925	1021	3111	3215	3533	3464	8592
maxval	260	261	286	334	3214	3243	3113	3115	8298
minval	265	254	273	355	3233	3147	3261	3116	8341

Table 1: Runtime performance (in microseconds) of reduction based Fortran intrinsics on the CPU and NPU's AIE array, operating on a one dimensional array of size 262144 elements.

the *result* variable. In this example it is the *sum* intrinsic call that is offloaded to the NPU by our approach.

We developed an AIE implementation for these reduction intrinsics which runs over four columns of the AIE array. A quarter of the array is held in each of the four memory tiles, and then the four compute tiles in that column each consume a quarter of their memory tile's data. The output of each AIE is a single summed number, and these 16 numbers are then read back by the CPU and summed together. Each AIE undertakes a vectorised reduction, operating on vectors of length 512 bits, and AIEs operate in batches of 16384 elements to decouple the data that is being processed from the 64KB of memory available on each compute tile.

Table 1 reports runtime performance of these reduction intrinsics on the CPU and NPU across data types. *maxval* and *minval* calculate the maximum and minimum array value. bfloat16 is not defined by the Fortran standard, so we developed a transformation to replace float32 in the IR with MLIR's bfloat16 type for those configurations. We also include a *conv-bfloat16* result in Table 1 for the NPU, where at runtime our XRT wrapper converts between float32 and bfloat16, with float32 on the CPU and bfloat16 running on the NPU.

It can be seen from Table 1 that, apart from the bfloat16 data type, the CPU very significantly outperforms the NPU at all data type configurations. The CPU performs poorly with bfloat16, most likely because it is being emulated in software. It can be seen that converting between bfloat16 and float32 in our XRT wrapper adds significant overhead. When exploring why the NPU was so much slower than the CPU we found that the majority of the runtime is constant regardless of the problem size and when run is first called significant setup time is incurred.

We therefore repeated our experiments, calling the intrinsic operation on the NPU multiple times, ignoring the initial time and recording the average runtime across the subsequent runs. This is reported in Table 2, where *total* is the total runtime, *xfer* is the data transfer component and *comp* is the compute time component. It can be seen that the runtime of subsequent runs on the NPU is significantly smaller than the initial run, and is competitive with, and sometimes outperforms, the CPU. It can also be seen that int32 and float32 are slower on the NPU than their int16 and bfloat16 counterparts, because the former are not supported directly in hardware by the AIE-ML.

We then explored offloading the *transpose* Fortran intrinsic onto the AIE array by leveraging AMD's *transpose-dma* example [1]. This example requires no involvement from the compute itself, but instead leverages the compute tile's data mover to undertake the transposition. We modified the code to instead use a memory tile, providing 512KB instead of 64KB in the compute tile. Given the

memory tile's fast memory and dedicated data movers, resulting in 30 GB/s bandwidth, it was our hypothesis that this could be beneficial compared to the CPU. Table 3 reports a performance comparison of undertaking transposition on the NPU's AIE array against the CPU for the *int32* data type. We report both the first execution runtime on the NPU and the average across subsequent runtimes. Performance on the NPU is fairly flat regardless of the array size, and in comparison performance grows in line with the data size on the CPU. For the largest array that can fit within the memory tile, the NPU outperforms the CPU, however the memory tile's 512KB memory tile is a significant limitation to the size of array that can be handled. Regardless, this again demonstrates the seamless use of the NPU without any modifications being required to the Fortran code.

The intrinsics considered so far in this section, although commonplace in Fortran codes, have been rather simple computationally. The Fortran standard defines a matrix multiplication, *matmul*, intrinsic which is much more computationally intensive and has also been optimised on the AIEs by AMD. We leveraged AMD's matrix multiplication example [1] that runs across the sixteen AIEs of the NPU, and integrated this with our compilation flow. Consequently, when the programmer calls the Fortran *matmul* intrinsic from their code this is then launched on the NPU.

Table 4 reports the performance of the matrix multiplication Fortran intrinsic for input input array sizes of 256x256 and 256x512, calculating a result array size of 256x512, across the CPU and NPU for different data types. It can be seen that the NPU outperforms the CPU for all configurations. The int32 and float32 calculations on the NPU are marked with an asterisk because this data type is used as the output, with the algorithm using the reduced precision counterpart (int16 and bfloat16 respectively) for inputs. The matrix multiplication kernel running on the NPU has been heavily optimised by AMD, taking advantage of vectorised multiply accumulate operations to gain best performance on the AIE architecture. This demonstrate the benefits of running suitable operations on the NPU, and although the AIE code itself is complicated and highly specialised this is all hidden from the Fortran programmer.

5 Conclusions and further work

Whilst the specialised computation provided by AMD's AI engines has the potential to deliver improved performance for appropriate workloads, especially as now these are integrated with AMD CPUs, it is not realistic to expect scientific programmers to have the required architecture specific knowledge and expertise. Consequently, in this paper we have explored an approach which enables seamless offloading of Fortran intrinsic calls onto AMD's Ryzen

Intrinsic	int16 runtime (us)			int32 runtime (us)			bfloat16 runtime (us)			float32 runtime (us)		
	total	xfer	comp	total	xfer	comp	total	xfer	comp	total	xfer	comp
sum	221	26	195	373	77	296	312	28	284	395	81	314
product	232	28	204	384	74	310	324	27	297	397	79	318
maxval	314	28	286	333	79	254	328	28	300	306	72	234
minval	312	28	284	327	78	249	335	28	307	316	74	242

Table 2: Runtime performance (in microseconds) of reduction based Fortran intrinsics for subsequent runs on the NPU's AIE array, operating on a one dimensional array of size 262144 elements. *xfer* is data transfer time, and *comp* compute time.

Array size	CPU runtime (us)	NPU first runtime (us)	NPU subsequent runtime (us)
64x64	11	358	194
128x128	151	446	235
256x256	203	418	240
512x256	576	440	230

Table 3: *transpose* Fortran intrinsic runtime with int32

Data type	CPU runtime (us)	NPU first runtime (us)	NPU subsequent runtime (us)
int16	5473	2572	1353
int32	14032	2635*	1503*
bfloat16	815194	2626	1357
float32	17566	3901*	1471*

Table 4: Runtime (in microseconds) of *matmul* Fortran intrinsic with a problem size of 256x256x512 elements

AI AIE array. Focussing on Fortran's intrinsics as a case study we have explored how the complexities of targetting an architecture such as the AIEs can be hidden by the compiler, and have demonstrated that by leveraging MLIR and building upon AMD's MLIR AIE support one is able to still deliver performance on the hardware. We developed an *xrt_wrapper* MLIR dialect to drive the NPU from the CPU, and generate IR for the AIEs based upon templates that stored in a library that are specialised by our approach for each instantiation.

We demonstrated that, whilst there is some initial setup overhead associated with running on the NPU, if intrinsics are repeatedly called from Fortran code, which is common in scientific computing, then this overhead can be ameliorated. This is especially important for fairly simple reduction based intrinsics, whereas the specialised nature of the NPU provides more clear benefits for array transposition at larger data sizes and especially for matrix multiplication.

In this paper we have focused on Fortran due to its popularity in scientific computing, but the linear algebra dialect is also leveraged by the ONNX dialect which is used extensively by ML frameworks, and future work will be to explore coupling and optimising our approach with those frameworks. Whilst we have focused on intrinsics in this paper, next steps will be to extend this to a wider range of algorithmic patterns such as stencils and generalised compute using common HPC programming models such as OpenMP and OpenACC. Work has already been undertaken mapping an MLIR stencil dialect to FPGAs [3], and we plan on extending this to also target AIEs.

We conclude that the offloading of Fortran intrinsics to the NPU in the AMD Ryzen AI CPU by the compiler demonstrates that MLIR is a potential game changer in enabling programmers to leverage the specialised compute provided by AIEs without requiring expert knowledge or effort.

Acknowledgments

References

[1] AMD Xilinx 2024. *MLIR-based AI Engine toolchain.* Retrieved Oct 8, 2024 from https://github.com/Xilinx/mlir-aie
[2] AMD Xilinx 2024. *Versal Adaptive SoC AIE-ML Architecture Manual.* Retrieved Oct 8, 2024 from https://docs.amd.com/r/en-US/am020-versal-aie-ml/Overview
[3] George Bisbas et al. 2024. A shared compilation stack for distributed-memory parallelism in stencil DSLs. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* 38–56.
[4] Nick Brown. 2023. Exploring the Versal AI engines for accelerating stencil-based atmospheric advection simulation. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 91–97.
[5] Nick Brown. 2024. Fully integrating the Flang Fortran compiler with standard MLIR. *arXiv preprint arXiv:2409.18824* (2024).
[6] Mathieu Fehr et al. 2022. IRDL: an IR definition language for SSA compilers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation.* 199–212.
[7] FIR 2024. *Design: Fortran IR.* Retrieved Aug 16, 2024 from https://flang.llvm.org/docs/FIRLangRef.html
[8] Flang 2024. *Flang Documentation.* Retrieved Aug 16, 2024 from https://flang.llvm.org/docs/
[9] Carsten Heinz, Jaco Hofmann, Jens Korinth, Lukas Sommer, Lukas Weber, and Andreas Koch. 2021. The TaPaSCo Open-Source Toolflow: for the Automated Composition of Task-Based Parallel Reconfigurable Computing Systems. *Journal of Signal Processing Systems* 93 (2021), 545–563.
[10] Carsten Heinz, Torben Kalkhof, Yannick Lavan, and Andreas Koch. 2024. TaPaS Co-AIE: An Open-Source Framework for Streaming-Based Heterogeneous Acceleration Using AMD AI Engines. In *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* IEEE, 155–161.
[11] HLFIR 2024. *High-Level Fortran IR (HLFIR).* Retrieved Aug 16, 2024 from https://flang.llvm.org/docs/HighLevelFIR.html
[12] Torben Kalkhof, Carsten Heinz, and Andreas Koch. 2024. Enabling FPGA and AI Engine Tasks in the HPX Programming Framework for Heterogeneous High-Performance Computing. In *International Symposium on Applied Reconfigurable Computing.* Springer, 75–89.
[13] Mark Klaisoongnoen, Nick Brown, Tim Dykes, Jessica R Jones, and Utz-Uwe Haus. 2024. Evaluating Versal AI Engines for option price discovery in market risk analysis. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays.* 176–182.
[14] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

[15] Maksim Levental, Arham Khan, Ryan Chard, Kyle Chard, Stephen Neuendorffer, and Ian Foster. 2024. An End-to-End Programming Model for AI Engine Architectures. In *Proceedings of the 14th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies*. 135–136.

[16] Rodriguez-Canal et al. 2023. Fortran High-Level Synthesis: Reducing the barriers to accelerating HPC codes on FPGAs. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 10–18.

[17] xDSL 2023. *A Python Compiler Design Toolkit*. Retrieved Aug 16, 2023 from https://github.com/xdslproject/xdsl

[18] Wenbo Zhang, Tianshuo Wang, Yiqi Liu, Yiming Li, and Zhenshan Bao. 2023. New Filter2D Accelerator on the Versal Platform Powered by the AI Engine. In *International Symposium on Advanced Parallel Processing Technologies*. Springer, 437–449.