OpenCL Code Generation for Low Energy Wide SIMD Architectures with Explicit Datapath

Dongrui She
Eindhoven University of Technology, the Netherlands
d.she@tue.nl

Yifan He
Eindhoven University of Technology, the Netherlands
y.he@tue.nl

Luc Waeijen
Eindhoven University of Technology, the Netherlands
l.j.w.waeijen@student.tue.nl

Henk Corporaal
Eindhoven University of Technology, the Netherlands
h.corporaal@tue.nl

***Abstract*—Energy efficiency is one of the most important aspects in designing embedded processors. The use of a wide SIMD processor architecture is a promising approach to build energy-efficient high performance embedded processors. In this paper, we propose a configurable wide SIMD architecture that utilizes explicit datapath to achieve high energy efficiency. To efficiently program the proposed architecture with a standard parallel programming language, we introduce a toolflow that can compile and map OpenCL programs onto it. The compiler in the proposed toolflow is able to analyze the static access patterns in OpenCL kernels and generate efficient mapping and code that utilizes the explicit datapath. Experimental results show that the proposed architecture is efficient. In a 128-PE processor, the proposed architecture is able to achieve over 200 times speed-up and reduce the energy consumption of register file and memory by over 90% compared to a RISC processor.**

***Index Terms*—Wide SIMD, Explicit Datapath, Low Power, OpenCL, Code Generation**

I. Introduction

Mobile systems like smart phones are becoming more and more important in daily life. The rapid development in embedded processors enables such devices to run high performance applications like wireless communication and high definition video codecs. However, energy efficiency is becoming the bottleneck in high performance embedded system design, especially for those ones that run on limited power sources like batteries. The single-instruction-multiple-data (SIMD) architecture is able to perform the same operation on multiple data simultaneously, thereby providing high computational throughput with low control overhead. Since many emerging embedded applications contain abundant data-level parallelism (DLP), using wide SIMD processors in such applications is a promising solution [1], [2].

In wide SIMD processors, the register files (RFs) are amount the most energy consuming components [1]. In this paper, we propose to use a wide SIMD architecture with explicit datapath to reduce the RF energy consumption. By using an explicit datapath that allows the software to directly control the data bypassing, the energy consumption of the processor is brought closer to the intrinsic energy consumption, i.e., the part of energy that is consumed by useful computation, instead of temporary data storage, data movement, and controlling.

An efficient compiler is a key to efficiently utilize the proposed architecture. It is well known that code generation for SIMD processors has always been one of the most difficult problems in compiler design. The open computing language (OpenCL) is a standard language for programming heterogeneous platforms. Initially it was designed for GPGPU architectures [3], [4], some of which are also wide SIMD processors. Therefore the OpenCL is suitable for programming low-energy SIMD processors. However, the proposed architecture poses additional challenges:

- compared to the flexible crossbar and network in GPGPU, the interconnect between PEs and the memory system in the proposed architecture only allow a limited form of communication between PEs, which makes efficient mapping of OpenCL kernels difficult;
- the bypassing in the PE datapath is directly controlled by the software, which means the compiler has to handle that in order to generate correct and efficient code.

We propose the design of a compiler that can analyze the OpenCL kernels that have statically analyzable memory accesses. The proposed compiler also utilizes the explicit datapath to generate highly energy efficient code.

Four kernels are tested on a 128-PE instance of the proposed architecture. The results are compared against a RISC processor and an SIMD processor without explicit bypassing. The average speed-up is $78\times$ compared to the RISC reference, which is the same as the non-explicit bypassing SIMD processor. And on the proposed architecture, the energy consumption of the registers and memories is reduced by 88.1% on average. While without explicit bypassing, the average improvement compared to RISC is only 50% due to the energy consumed by redundant accesses to the vector register file. The key contributions of this paper are:

- We propose a wide SIMD processor architecture that can efficiently support the mapping of OpenCL kernels.
- A compiler for the proposed architecture is designed. The compiler compiles OpenCL program and optimizes memory mapping for the proposed architecture.
- Detailed experiments are carried out. The results show that the proposed architecture and compiler is able to achieve substantial improvement in both performance and energy consumption.

The remainder of this paper proceeds as follows: Section II introduces the background information of explicit datapath architectures and the OpenCL programming language. The proposed architecture is described in Section III. Section IV introduces the compiler design and the mapping of OpenCL kernels in the proposed architecture. Experimental results that show the effectiveness of the proposed design are given in Section V. Related work is discussed in Section VI. Finally, Section VII concludes our findings and discusses future work.

II. Background

In this section the concepts that are essential to this work are introduced. Section II-A describes the idea of explicit datapath. The basic knowledge of the OpenCL language is given in Section II-B.

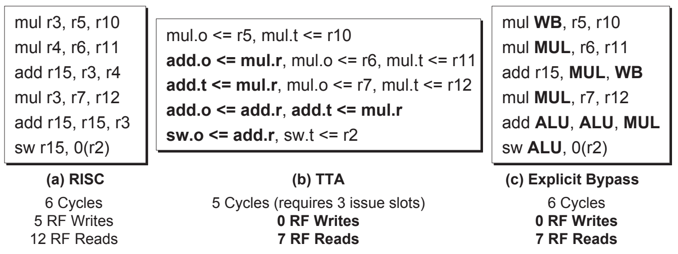
A. Processors with Explicit Datapath

The key idea of explicit datapath architectures is to expose more details of the datapath to the software, thereby enabling fine-grained control over the datapath in the software. By having fine-grained control, a considerable amount of redundant data movement can be eliminated, which can potentially result in improvement in performance and energy efficiency.

The transport-triggered architecture (TTA) is a prime example of explicit datapath architectures [5], [6]. Fig. 1(b) shows an example sequence of TTA instruction that performs the same operations as the RISC instructions in Fig. 1(a). In a TTA, the software controls data movement, and operations are *side-effects* of the data movements. By allowing software to have full control every data movement in the datapath, TTA is able to reduce the register file port requirements dramatically and improve performance. Recent studies also exploit the explicit datapath in TTAs for building energy efficient processors [7], [8], [9]. Two common problems can hurt the energy efficiency of TTA-based processors: *i*) low code density, and *ii*) more flexible, therefore more complex interconnect between FU/RF input and output ports. The code density problem can be mitigated by code compression [10] and micro-architecture modification [7]. The interconnect overhead in TTAs can be reduced by interconnect reduction, which is particularly effective in application specific processor design.

Using explicit bypassing is an alternative of building explicit datapath architecture [11], [12], [13], [14]. The bypassing (forwarding) network in a conventional processors is used to reduce data hazard caused by the pipelining. In an explicit bypassing architecture, this network is exposed to the software. Fig. 1(c) shows a code fragment of explicit software bypass that performs the same operation as Fig. 1(a). In explicit bypass architectures, the ISAs are typically similar to conventional processors. The main difference compared to conventional architectures is that part of the internal pipeline state is exposed to the software, thereby enabling dramatic decrease of redundant register file traffic. Compared to TTAs, explicit bypass architectures offer less flexibility in datapath control. On the other hand, such architectures have less overhead in control and interconnect.

An explicit datapath is particularly interesting for a wide SIMD architecture. The vector register files often consume considerable amount of energy in SIMD processors [1]. Meanwhile the potential control overhead can be amortized by the large number of processing elements (PEs). In this work, explicit bypassing is used to build the PEs of the proposed SIMD architecture.



```
mul r3, r5, r10
mul r4, r6, r11
add r15, r3, r4
mul r3, r7, r12
add r15, r15, r3
sw r15, 0(r2)
```

(a) RISC
6 Cycles
5 RF Writes
12 RF Reads

```
mul.o <= r5, mul.t <= r10
add.o <= mul.r, mul.o <= r6, mul.t <= r11
add.t <= mul.r, mul.o <= r7, mul.t <= r12
add.o <= add.r, add.t <= mul.r
sw.o <= add.r, sw.t <= r2
```

(b) TTA
5 Cycles (requires 3 issue slots)
0 RF Writes
7 RF Reads

```
mul WB, r5, r10
mul MUL, r6, r11
add r15, MUL, WB
mul MUL, r7, r12
add ALU, ALU, MUL
sw ALU, 0(r2)
```

(c) Explicit Bypass
6 Cycles
0 RF Writes
7 RF Reads

Fig. 1. Comparison the codes of RISC, TTA and explicit bypass

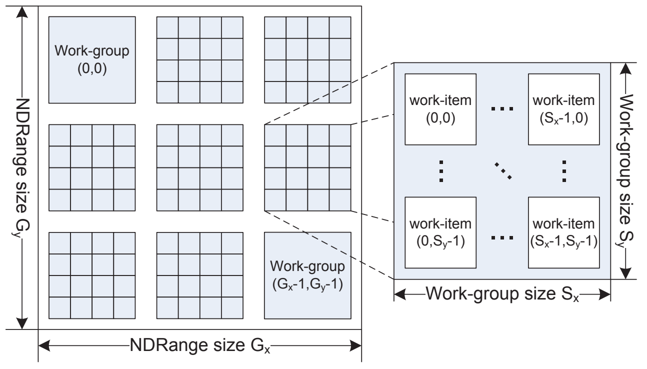


Fig. 2. The index space of a 2-D OpenCL kernel

B. OpenCL Parallel Programming Language

OpenCL (Open Computing Language) is an open standard for parallel programming across different types of devices, including CPUs, GPUs and accelerators [15]. The OpenCL standard defines:

- a C-based language called OpenCL C that used to define *kernels* for performing computation on *compute devices*;
- a set of APIs in standard C for invoking the kernels from the *host*.

In an OpenCL kernel, the workload is divided into *work-groups*. Each work-group consists of a number of *work-items*. Fig. 2 illustrates the index space of an OpenCL kernel. In OpenCL kernel semantics, every work-item executes the kernel function independent of other work-items. Different work-items can only synchronize by calling synchronization functions explicitly. So different work-items of the same kernel can be executed in parallel between explicit synchronization points. This model is ideal for wide SIMD architectures because: *i*) work-items of the same kernels execute the same instruction sequence, which is easy to fit in SIMD semantics; *ii*) the implicit independence of work-items gives the compilers more freedom to map and schedule them on SIMD processors.

The conceptual device architecture in OpenCL is shown in Fig. 3. There are four different address spaces, namely, private, local, global and constant. Each work-item is mapped onto a processing element (PE) and each work-group is mapped onto

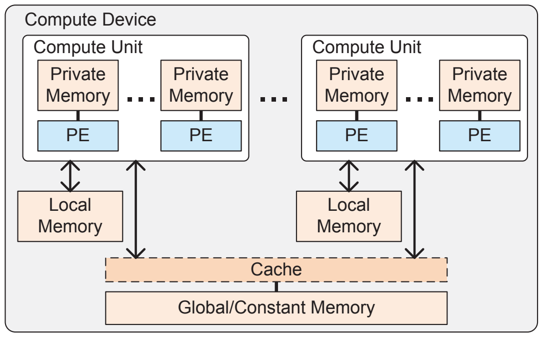


Fig. 3. Conceptual OpenCL device architecture

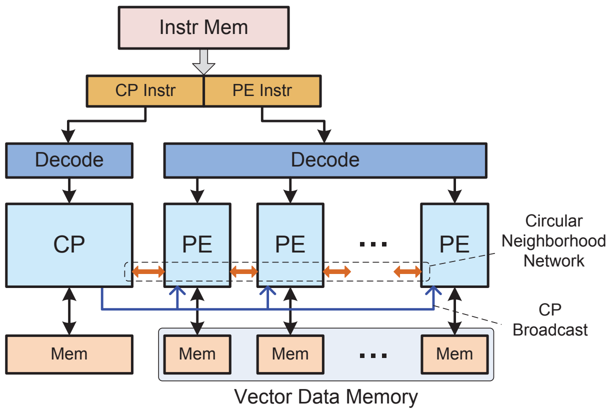


Fig. 4. Proposed wide-SIMD processor architecture

a compute unit. The different address spaces make the analysis and mapping of communication between work-items easier for wide-SIMD architectures.

The two most important aspects of mapping OpenCL kernels onto a processor architecture are:

- Map and schedule work-items on the PEs of the target architecture;
- Map the different address spaces onto the memory hierarchy of the target architecture.

III. Wide-SIMD Architecture with Explicit Datapath

The wide SIMD processor architecture used in this work is based on the one in [16]. Some modifications are made to improve the support the mapping of OpenCL programs. The proposed processor architecture consists of two parts, a control processor (CP) and a wide one dimensional (1-D) array of processing elements (PEs), which run in lock-step. It results in a VLIW processor with one scalar issue slot for the CP and one vector issue slot for the PE array. Fig. 4 depicts the proposed architecture.

The ISA of both the CP and the PE is based on a 24-bit RISC-like ISA similar to the one used in [13]. Table I shows the key features of the baseline ISA.

To adapt to the proposed SIMD architecture, the following modifications are made to the baseline ISA:

- Only the CP can execute control instructions.

TABLE I
Key features of the baseline ISA

Instruction width	24 bits
Data width	32 bits or 16 bits
Pipeline stages	4 or 5
Register file	32b/16b×32, 2R1W
Opcode	6 bits
Immediate	8 bits
Branch immediate (CP)	16 bits
Branch delay slot	1

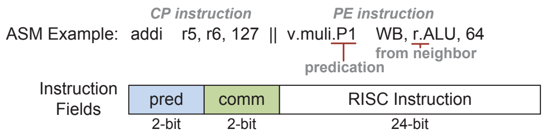


Fig. 5. CP and PE instruction format

- Two bits are added to the instruction format to encode the communication. The first source operand of each instruction may come from one of: *i*) local RF/bypass, *ii*) left neighbor, *iii*) right neighbor, *iv*) CP broadcasting. More details are given in Section III-B.
- A two-entry predicate register file is introduced. Extra two bits are used to encode whether each instruction is predicated by 0, 1 or both predicate registers. This is essential for mapping OpenCL kernel efficiently, as it simplifies the control flow mapping.

The instruction format of the proposed architecture is shown in Fig. 5. The size of a 2-issue VLIW instruction packet (1 CP instruction + 1 PE instruction) is 56 bits.

A. *Explicit Datapath*

In the proposed architecture, the datapath of the PE uses explicit bypassing. The datapaths of the 4-stage and 5-stage PE are show in Fig. 6. Compared to conventional architectures, instructions in the proposed architecture have more control over input operands and destinations:

- An instruction directly specifies whether an input operand is from a register RF or one of the bypassing sources.
- Each functional unit (FU) in the datapath has separate input registers that ensures the result of the FU remains stable until the next operation is issued to the FU, resulting in more bypassing opportunities;
- Each instruction can control whether the result needs write-back. Three options are available:
  - No write-back: the result is only available at the FU output;
  - To WB stage: the result is written to the pipeline register in the write-back stage, but not to the RF;
  - To RF: the result is stored into the RF.

To use explicit bypass without changing the instruction format, part of the RF index space is used for the bypassing sources. As a result, the number of registers in the RF is reduced from 32 to 28 (4-stage) or 27 (5-stage). The impact of a smaller RF is mitigated by the fact that there is no need to allocate registers for short-live variables in many cases, as

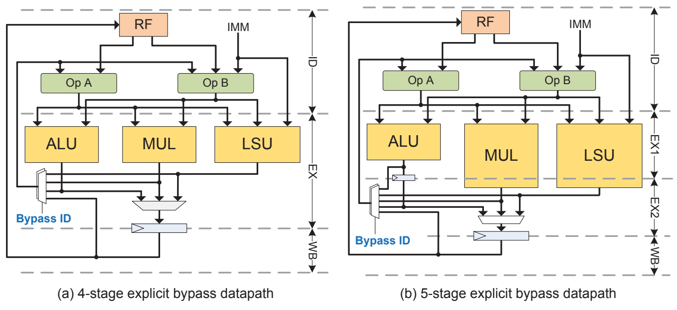

RF
IMM
Op A
Op B
ALU
MUL
LSU
Bypass ID
ID
EX
WB
EX1
EX2
(a) 4-stage explicit bypass datapath
(b) 5-stage explicit bypass datapath

Fig. 6. Datapath with explicit bypass

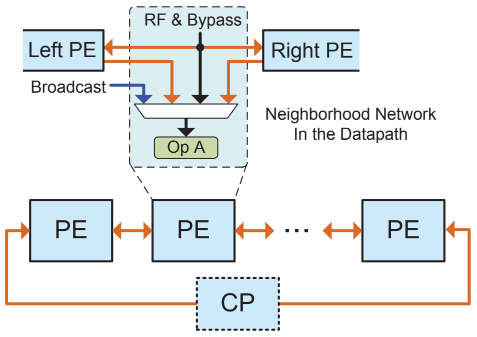


Fig. 7. Circular neighborhood communication network

shown in the example in Fig. 1(c).

B. Circular Neighborhood Communication Network

Unlike SIMD architectures with small vectors, a fully connected shuffling network is not scalable in a wide SIMD processor. In the proposed architecture, a one dimensional (1-D) neighborhood network is used. Fig. 7 shows the architecture of the network. Each PE can only communicate with its left and right neighbors. The network is circular, i.e., the first and last PEs can be neighbors. The connection between the first PE and the last PE does not introduce extra long wires, since in physical layout the PEs can be placed in a circular manner. The PE accesses the network in the decode stage where source operands are collected for an instruction. The first operand of each instruction comes from either the local RF/Bypass or the communication network. Two bits in each instruction are used to indicate which source it is using.

The CP can be an extra node in the PE communication network, allowing data exchange between the scalar datapath and the vector datapath. In addition, the CP is able to broadcast data to all PEs, allowing the CP to perform calculations that are common to all PEs, which could be more energy efficient.

C. Accelerator Architecture

Fig. 8 depicts a generic architectural template for using the proposed architecture as an accelerator. The slave interface allows the host to have direct access to all the memories of the accelerator. For more efficient data transfer, a direct-memory-access (DMA) controller is used to move data between external memory and the memories of the accelerator.

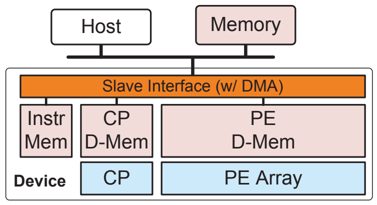


Fig. 8. Accelerator based on the proposed architecture

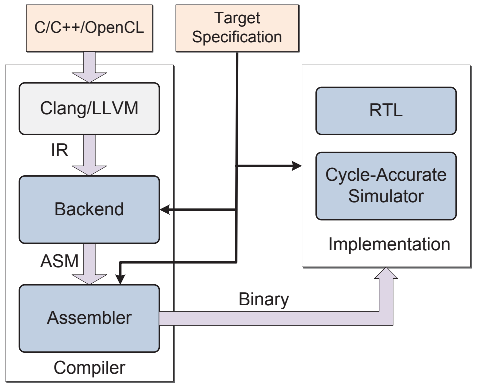
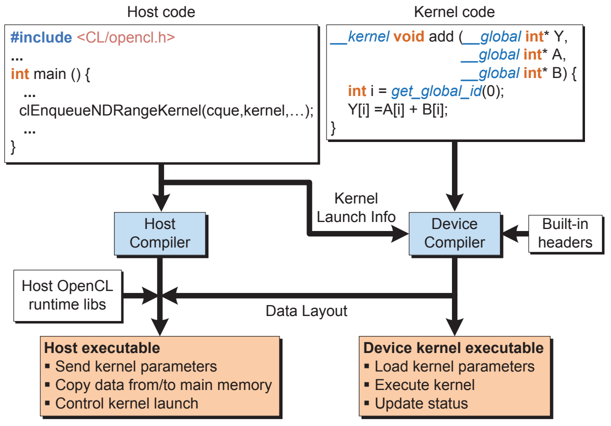


Fig. 9. Proposed toolflow

IV. OpenCL Code Generation

A retargetable compiler is designed for the proposed SIMD processor architecture. Fig. 9 shows the structure of the toolflow for the proposed architecture. The frontend of the compiler is based on the open-source LLVM compiler framework [17]. The frontend produces a low-level intermediate representation (IR). The backend processes the IR and the target specification and produces the final code. Fig. 10 shows the process to compile an OpenCL program that runs on the platform depicted in Fig. 8. The host code is compiled by the native compiler of the host processor. And the device code is compiled by the compiler shown in Fig. 9.

The remainder of this section gives more details about the compiler backend. Section IV-A introduce how a OpenCL program is mapped onto the proposed architecture. The analysis and optimizations for interleaved work-item and data mapping


Host code

```
#include <CL/opencl.h>
...
int main () {
  ...
 clEnqueueNDRangeKernel(cque,kernel,…);
  ...
}
```

Kernel code

```
__kernel void add (__global int* Y,
                   __global int* A,
                   __global int* B) {
  int i = get_global_id(0);
  Y[i] =A[i] + B[i];
}
```

Fig. 10. Compilation process for OpenCL program

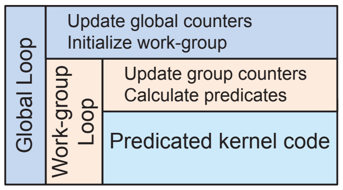


Fig. 11. Converted OpenCL kernel, two loop-nests are inserted

TABLE II
MAPPING OF DIFFERENT OPENCL ADDRESS SPACES

Address Space	Mapping
Global/Local	Data memory of the whole PE array
Private	Memory bank of each PE
Constant	CP or PE memory, depending on access pattern

are described in Section IV-B. Lastly, Section IV-C shows how the explicit data in the PE array is utilized.

A. Basic OpenCL Kernel Mapping

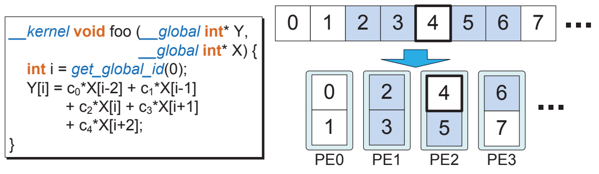
Explicit loops are created on the CP to execute different work-items. Different work-groups are executed sequentially. The work-items in a work-group are mapped to the PE array. If the launch parameters and memory accesses in the kernel can be statically analyzed, the mapping of the kernel is generated by the method described in Section IV-B. For work-group with size that is not aligned to the number of PEs, predication is used to guard the execution. Fig. 11 illustrates how the control flow of the kernel is converted. For work-group synchronization barriers, this work uses a similar approach as [18]. The work-group loop is split at each barrier, which ensures all work-items finish the work prior to the barrier before continuing.

Table II shows the mapping of different address spaces in OpenCL to the memory hierarchy of the proposed architecture. The global and local address spaces are both mapped to the vector memory of the PE array. A series of data shifting via the neighborhood network is inserted if the data needed by a PE is not in the memory bank of that PE. The private memory space is only accessible by a work-item. Therefore it is natural to map data in private memory to the memory bank of each PE. The data in the constant memory can be mapped to the memory of the CP if all PEs access the same data in each loop iteration. Otherwise it is mapped as constant data in the memory bank of PE.

To generate code for a complete OpenCL program for the system shown in Fig. 8, the proposed toolflow requires the complete sources of both the host and the device to be available at compile time. The host processor controls the kernel launching. For each kernel, the host processor sends the launch parameters to the accelerator. The input data and instructions are copied to the local memory of the accelerator by the DMA controller. After the kernel execution, the output data is copied out by the DMA controller.

B. Interleaved Work-Item Mapping and Data Layout

For generic kernels, the work-items are mapped onto the PE array linearly, i.e., the i-th work-item is mapped to the $i\%N$-th PE, where N is the number of PEs. If there are more than one dimension in the index space, only the first dimension is mapped to the vector, and the others are controlled by CP loops. In the PE array of the proposed architecture, each PE can only communicate with its left and right neighbors. Therefore long distance communication is not efficient. In this work, *interleaved* mapping and data layout are used to reduce the communication distance.



```
__kernel void foo (__global int* Y,
                   __global int* X) {
  int i = get_global_id(0);
  Y[i] = c0*X[i-2] + c1*X[i-1]
         + c2*X[i] + c3*X[i+1]
         + c4*X[i+2];
}
```

Fig. 12. Example of interleaved mapping with factor of 2

Fig. 12 shows an example of the interleaved mapping. The index in Fig. 12 represents both the work-item and the linear address of the data. Each work-item needs to access data in a window of size 5, e.g., work-item 4 needs to access data in address 2 to 6. If a linear mapping is used, each work-item needs to communicate with PEs two steps away, which is less efficient in the proposed architecture. By using a mapping with interleaving factor of 2, as shown in Fig. 12, the maximum communication distance is 1 instead of 2. Each OpenCL memory buffer object is analyzed by its access patterns. If there are different interleaving factors for multiple accesses (from the same or different kernels), the biggest one is used as the actual interleaving factor. The interleaving information is also used to generate the host code for proper data transfer between the system memory and the local memory of the accelerator. The host processor programs the DMA controller according to the interleaving factors determined by the device compiler. In the current implementation, kernels have to be compiled *off-line* in order to use the interleaved mapping.

When a kernel is mapped with an interleaving factor $N > 1$, the work-group loop has to be unrolled N times in order to handle the irregular communication pattern, which may introduce energy overhead. For example, in Fig. 12, work-item 4 needs two samples from left and one from right, while work-item 5 needs one from left and two from right. Therefore they need different instructions and the kernel has to be unrolled.

The limitation of the interleaved mapping and layout in this work is that it requires: *i*) all address expressions for the global and local memory can be analyzed statically; *ii*) the kernel launch parameters are compile-time constants, or chosen from a set of compile-time constants; If the compiler fails to analyze the communication parameters statically, generic communication is required. A generic communication loop for store is shown in Algorithm 1. The upper bound of the distance U is the maximum distance in number of hops in the communication network, which is the number of PEs in the worst case. The loop for load is similar. Some optimizations are possible, e.g., if the compiler knows that the communication is one-to-one, the result can be put in a register instead of the memory.

Algorithm 1: Generic PE store with communication

```
Input : Distance upper bound $U$, shift direction $d$, source vector $S$,
        destination address vector $D$
# No assumption about which PE an address in $D$ is
# mapped to. Loop through all PEs and check dynamically
for $i$ =0 to $U$ do
    for Each PE $i$ do
        if $D[i]$ is in PE $i$ then
            Store $S[i]$ in the memory of PE $i$
        end
    end
    Shift $S$ by 1 step in $d$
    Shift $D$ by 1 step in $d$
end
```

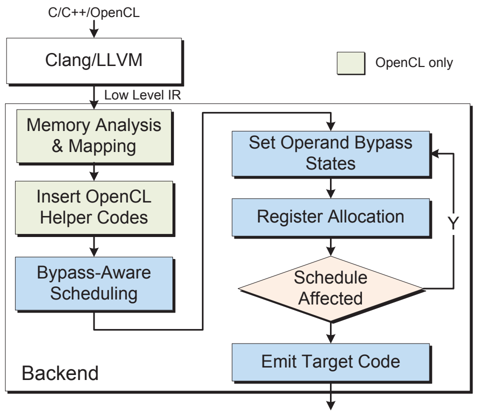


Fig. 13. The work-flow of the compiler backend

C. Code Generation for Explicit Datapath

To generate correct and energy efficient code, the compiler has to be aware of the explicit bypassing. Fig. 13 shows the flow of the compiler backend in this work.

A pre-register-allocation basic-block level bypass-aware list scheduling algorithm is used. The amount of bypassing of each instruction is used as the priority in selecting candidates from the ready list. The scheduler schedules instruction in bottom-up fashion, i.e., an instruction becomes ready when all of other instructions depend on it are scheduled. In the bottom-up scheduling, the scheduler knows precisely whether the result of the instruction to be scheduled needs to be written back to the register file.

The operand bypass state is set after the scheduling. However, the schedule may change in register allocation pass if there is spilling. In that case, the operand bypass state initialization and the register allocation need to be run again. This process is illustrated in the loop in Fig. 13. Though in practice the loop usually terminates quickly, because spill and reload codes on the proposed architecture do not need extra registers thanks to the explicit bypassing.

V. Experimental Results

In this work, the metrics used to evaluate the proposed architecture are performance and energy consumption. The four kernels used for evaluation are listed in Table III. The experimental setup is described in Section V-A. The performance and energy results are shown in Section V-B and Section V-C, respectively.

TABLE III
Kernel description

Kernel	Description
MAdd	Matrix addition
FIR	5-tap FIR filter on 1-D data stream
Sobel	3x3 Sobel edge detection filter
Transpose	Square matrix transpose

TABLE IV
Configuration of the target architecture

Data width	32 bits
Pipeline stages	4
Number of PEs	128
Instruction memory	56b × 1k
PE data memory	1k entries

A. Experimental Setup

To evaluate the performance and energy efficiency of the proposed design, the kernels in Table III are tested on one 128-PE SIMD processor with automatic bypassing (*SIMD*) and one 128-PE SIMD processor with explicit bypassing (*SIMD-b*). The key parameters of the two processors are listed in Table IV.

The kernel codes written in OpenCL are compiled by the proposed compiler off-line, i.e., at compile time. The generated binaries are executed on the cycle-accurate simulator for the SIMD processor to collect statistics. Due to the lack of implementation of a complete platform as shown in Fig. 8, the host processor is emulated by a test driver program. Therefore, parts of the host overhead, e.g., the cost of re-organizing data layout, are not reflected in the results in this section. Sequential codes written in C are compiled and run on a 4-stage 32-bit RISC processor as the reference (*RISC*). The RISC processor has 4kB instruction memory and 16kB data memory.

B. Performance

Fig. 14 shows the normalized cycle count of each kernel on different processors. The average speed-up of the proposed architecture with regard to the RISC reference is $78\times$. The use of explicit bypassing does not have any visible impact on the performance. For kernels with regular memory access patterns, including the MAdd and FIR, The speed-up compared to RISC processor is ideal. In the FIR kernel, the speed-up is larger than the number of PEs, due to the instruction level parallelism exploit by the dual issue of the proposed architecture. It is also shown that by using the interleaved mapping described in Section IV-B, the FIR kernel gains extra 25% in performance.

The performance of the matrix transpose kernel is considerably worse compared to other kernels. The main reason is that in matrix transpose, long distance communication is required. And since only neighborhood communication is possible, such kernel is not efficient on the proposed architecture.

C. Energy

In this work, we focus on the energy consumption of the memory and register file energy. For each processor, the energy consumption of the following components is taken into account:

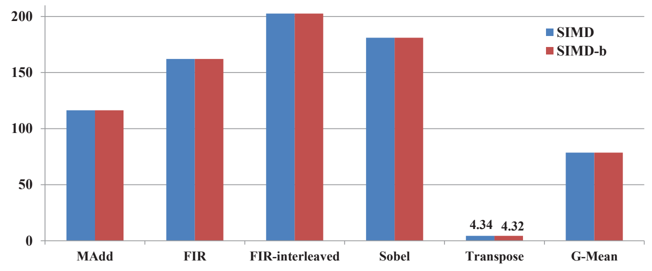


Fig. 14. Speed-up compared to RISC

TABLE V
ENERGY CONSUMPTION OF DIFFERENT DATA ACCESSES

	32b×32 2R1W RF		4kB Memory	16kB Memory	14kB Memory
	Read	Write	32-bit	32-bit	56-bit
Access Energy (pJ)	0.76	1.37	2.02	2.92	3.37

- Register file (*Reg*, a vector access is treated as 128 scalar accesses, which is in line with the implementation);
- Data memory (*DMem*, a vector access is treated as 128 scalar accesses, since the memory bank of each PE is separately addressable);
- Instruction memory (*IMem*).

These components consume considerable portion of energy in SIMD processors [1], therefore they are good indicator of the energy consumption of the whole processor. The technology we used is 40nm low-power CMOS. The energy consumption of the memory is estimated by CACTI [19]. The RF access energy is derived by synthesizing the RTL design with 40nm TSMC low-power library, extracting the physical information, and estimating the average toggle rate of each port by performing 2048 random accesses. Table V shows the energy consumption of different types of accesses.

Fig. 15 shows the number of register accesses. For the SIMD processor with automatic bypassing, the reduction in register access comes mostly from the fact that the control-related instructions are run on the CP, which greatly reducing the number of register accesses in these instructions. It is clear that the explicit bypassing has dramatic impact on the number of register accesses. In particular, almost all register accesses are eliminated in the MAdd and FIR. In the transpose kernel, the SIMD processor with automatic bypassing has much more register accesses then RISC due to the communication. In contrast, the processor with explicit bypassing is able to eliminate all register accesses in the communication, resulting in over 70% decrease in number of accesses.

Fig. 16 shows the normalized energy results. Both SIMD processors benefits from the reduction in instruction memory, which is a natural advantage of SIMD architectures. However, it is also clear that the reduction in the register access has noticeable impact in the energy consumption. Compared to RISC, the SIMD processor with explicit bypassing reduced the energy consumption by 88.1%, which is 78.4% less than the one with automatic bypassing. Even in the matrix transpose kernel where it is inefficient on the proposed architecture, the energy consumption is still reduced by over 50%, whereas the one with automatic consumes 4.05 times more energy than the

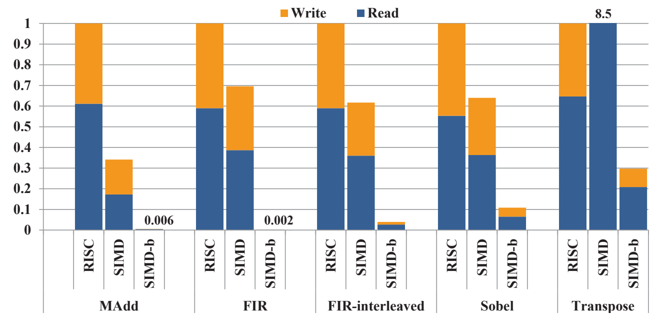


Fig. 15. Number of register accesses (normalized to RISC)

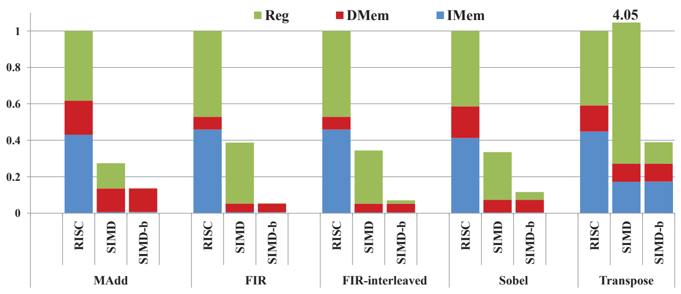


Fig. 16. Energy consumption (normalized to RISC)

RISC reference.

As mentioned in Section IV-B, the interleaved mapping introduces a small overhead in the number of register access. The reason is that in the unrolled kernel, samples need to be stored in register for the following work-item, preventing the elimination of write-back. It also results in small increase in energy consumption.

In all, the proposed architecture and compiler are able to achieve substantial improvement in both performance and energy consumption.

VI. RELATED WORK

Wide SIMD architectures are used in many embedded processors. The Xetal from NXP [20] is an SIMD processor with 320 PEs that is designed for smart camera data processing. The PEs in Xetal are connected by neighborhood network. However, due to the lack of register file, the energy consumption of the vector memory is high. Y. He et al addressed this problem in Xetal-Pro, by introducing extra level of memory, as well as aggressive voltage scaling, resulting in a much more efficient architecture [2]. The IMAP from NEC [21] is another example of wide SIMD processor. The IMAP has 128 PEs connected with a ring network. A key difference in IMAP compared to Xetal is that it has independent address generation for each PE. While the memory is more complex in such configuration, it also results in much better programmability. Woh et al. proposed AnySP, a wide SIMD targeting wireless and multimedia applications [1]. The PE interconnect in AnySP is a reconfigurable RAM-based crossbar, which is more flexible compared to Xetal and IMAP. The energy of the vector register file in AnySP is reduced by introducing an extra 4-entry small register file. In this work, the proposed architectures is similar to the Xetal-Pro [2]. The main difference is the the proposed architecture uses per-PE register file and index addressing, and a PE datapath with explicit bypassing. Results show that the proposed architecture achieved high energy-efficiency.

Programming wide SIMD architectures has always been difficult. IMAP uses a dedicated C dialect called *one-dimension C* (1DC) to develop data-parallel processing programs [21]. Languages like 1DC can be fine tuned for the target architecture and is used by similar architectures such as Xetal. But they lack portability and are not compatibility with standard languages. OpenCL is a standard parallel programming language for heterogeneous platforms [15]. It is initially design for GPGPU architectures [3], [4]. And it can also be mapped to general purpose CPUs efficiently [22]. Recent studies attempt to use OpenCL for more diverging target architectures, such as FPGA [23] and ASIP [18]. In this work, in an effort to support standard programming language to low-energy wide SIMD architecture, we presented the design of a compiler that is capable of compiling OpenCL program on the proposed wide SIMD architecture.

VII. Conclusions and Future Work

In this paper, an energy-efficient wide SIMD processor architecture is introduced. The proposed architecture has a scalable neighborhood communication network and the datapath in PEs uses explicit bypassing. An OpenCL compiler design is proposed for the target architecture. The proposed compiler is able to analyze OpenCL kernels and generate energy efficient code by optimizing memory mapping and utilizing explicit bypassing. Experimental results for a 128-PE instance show that the combination of the proposed architecture and compiler is efficient: compared to the RISC reference, the average speed-up is $78\times$, and the energy consumption of the registers and memories is reduced by 88.1% on average.

Future work includes a more sophisticated analysis and optimization of the memory layout and work-item scheduling. In particular, handling less static address expressions would be useful for generalizing the proposed architecture. It is also very interesting to perform software-hardware exploration for improving the inter-PE communication in an energy efficient way such that kernels with irregular communication pattern are easier and more efficient. In addition, further changes in the architecture, e.g., clustering PE memory banks to reduce memory energy, also require the adaption of the compiler.

Acknowledgments

This work is supported by the Dutch Technology Foundation STW, project NEST 10346, and the Ministry of Economic Affairs of the Netherlands, project EVA PID07121.

References

[1] M. Woh et al., "AnySP: anytime anywhere anyway signal processing," in *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*, 2009, pp. 128–139.

[2] Y. He et al., "Xetal-Pro: An Ultra-Low Energy and High Throughput SIMD Processor," in *Proceedings of the 47th Annual Design Automation Conference (DAC '10)*, 2010, pp. 543–548.

[3] AMD. AMD OpenCL Zone. [Online]. Available: http://developer.amd.com/resources/heterogeneous-computing/opencl-zone

[4] C. Wittenbrink et al., "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[5] H. Corporaal, *Microprocessor Architectures: From VLIW to TTA*. Wiley, 1998.

[6] O. Esko et al., "Customized exposed datapath soft-core design flow with compiler support," in *Proceedings of 20th International Conference on Field Programmable Logic and Applications*, 2010, pp. 217–222.

[7] Y. He et al., "MOVE-Pro: a low power and high code density tta architecture," in *Proceedings of the 11th International Conference on Embedded Computer Systems (SAMOS-XI)*, 2011, pp. 294–301.

[8] D. She et al., "Scheduling for register file energy minimization in explicit datapath architectures," in *Design, Automation Test in Europe Conference Exhibition, 2012 (DATE '12)*. EDAA, 2012, pp. 388–393.

[9] V. Guzma et al., "Reducing processor energy consumption by compiler optimization," in *IEEE Workshop on Signal Processing Systems (SiPS)*, 2009, pp. 63–68.

[10] J. Heikkinen et al., "Dictionary-based program compression on TTAs: effects on area and power consumption," in *Proceedings of the 2005 IEEE Workshop on Signal Processing Systems Design and Implementation*, 2005, pp. 479–484.

[11] J. Balfour et al., "An energy-efficient processor architecture for embedded systems," *Computer Architecture Letters*, vol. 7, no. 1, pp. 29–32, 2007.

[12] ——, "Operand registers and explicit operand forwarding," *Computer Architecture Letters*, vol. 8, no. 2, pp. 60–63, 2009.

[13] D. She et al., "Energy efficient special instruction support in an embedded processor with compact isa," in *Proceedings of the 2012 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES '12)*. ACM, 2012, pp. 131–140.

[14] I. Finlayson et al., "An overview of static pipelining," *Computer Architecture Letters*, vol. 11, no. 1, pp. 17–20, 2012.

[15] Khronos OpenCL Working Group, "The OpenCL Specification, version 1.2," 2012, http://www.khronos.org/registry/cl/.

[16] L. Waeijen et al., "SIMD made explicit," in *Proceedings of the 13th International Conference on Embedded Computer Systems (SAMOS-XIII)*, 2013.

[17] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004, pp. 75–86.

[18] P. Jääskeläinen et al., "OpenCL-based design methodology for application-specific processors," in *Proceedings of the 10th International Conference on Embedded Computer Systems (SAMOS-X)*, 2010, pp. 223–230.

[19] CACTI, "cacti 5.3, rev 174," http://quid.hpl.hp.com:9081/cacti/.

[20] A. Abbo et al., "Xetal-II: a 107 GOPS, 600 mW massively parallel processor for video scene analysis," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 192–201, 2008.

[21] S. Kyo and S. Okazaki, "IMAPCAR: A 100 GOPS In-Vehicle Vision Processor Based on 128 Ring Connected Four-Way VLIW Processing Elements," *Journal of Signal Processing Systems*, pp. 1–12, 2008.

[22] R. Karrenberg and S. Hack, "Improving performance of OpenCL on CPUs," in *Proceedings of the 21st International Conference on Compiler Construction (CC '12)*. Springer-Verlag, 2012, pp. 1–20.

[23] M. Owaida et al., "Synthesis of platform architectures from OpenCL programs," in *Proceedings of the 19th International Symposium on Field Programmable Custom Computing Machines (FCCM '11)*. IEEE, 2011, pp. 186–193.