OpenCL-based Design Methodology for Application-Specific Processors

Pekka O. Jaasklainen, Carlos S de La Lama, Pablo Huerta and Jarmo H. Takala

Presented by Benjamin Turk

What is this paper about?

- The paper shows a procedure on how to map the relatively simple OCL kernel to a pipelined hardware design.
- They generate the hardware Verilog in a fully automatized way.
- They perform experiments while mapping the kernel and share the results.

Outline

1. Brief Intro to OpenCL(OCL)
2. Application Specific Processors (ASP)
3. Practical Issues in compiling OCL
4. Experiments
5. Conclusion

1.Brief Intro to OpenCL(OCL)

- What is OpenCL?
- What makes OCL an attractive?
- Features

What is OpenCL?

- A standard for programming heterogeneous multiprocessor platforms.

1024 x 1024 image:
problem dimensions:
1024 x 1024 = 1 kernel
execution per pixel:
1,048,576 total executions

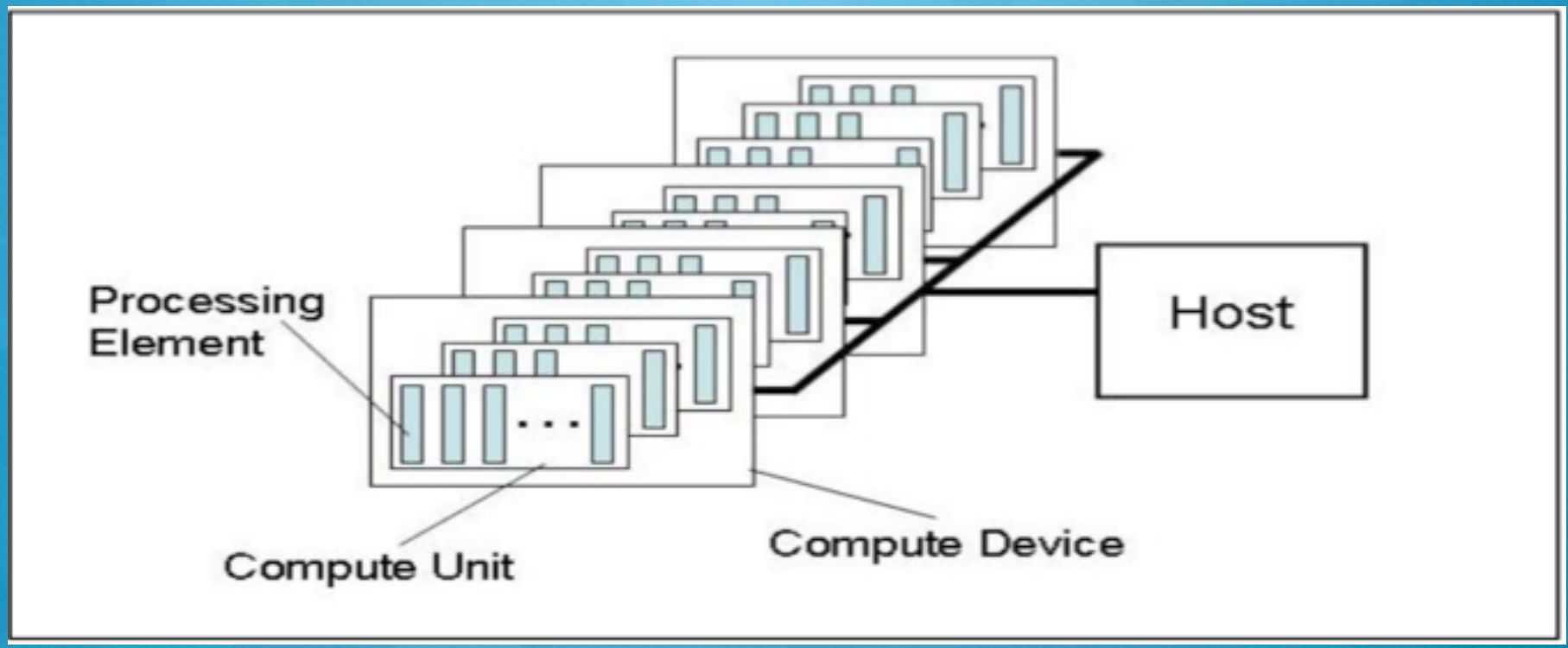
Scalar

```
void
scalar_mul(int n,
           const float *a,
           const float *b,
           float *result)
{
  int i;
  for (i=0; i<n; i++)
    result[i] = a[i] * b[i];
}
```

Data-Parallel

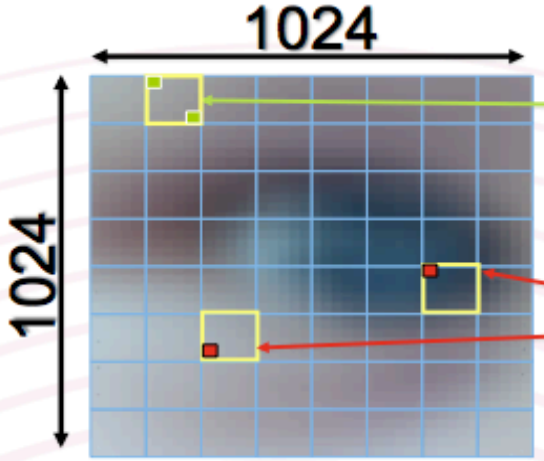
```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *result)
{
  int id = get_global_id(0);
  result[id] = a[id] * b[id];
}
// execute dp_mul over "n" work-items
```

OpenCL Platform Model



An N-dimension of work-items

- **Global Dimensions: 1024 x 1024 (whole problem space)**
- **Local Dimensions: 128 x 128 (work group … executes together)**



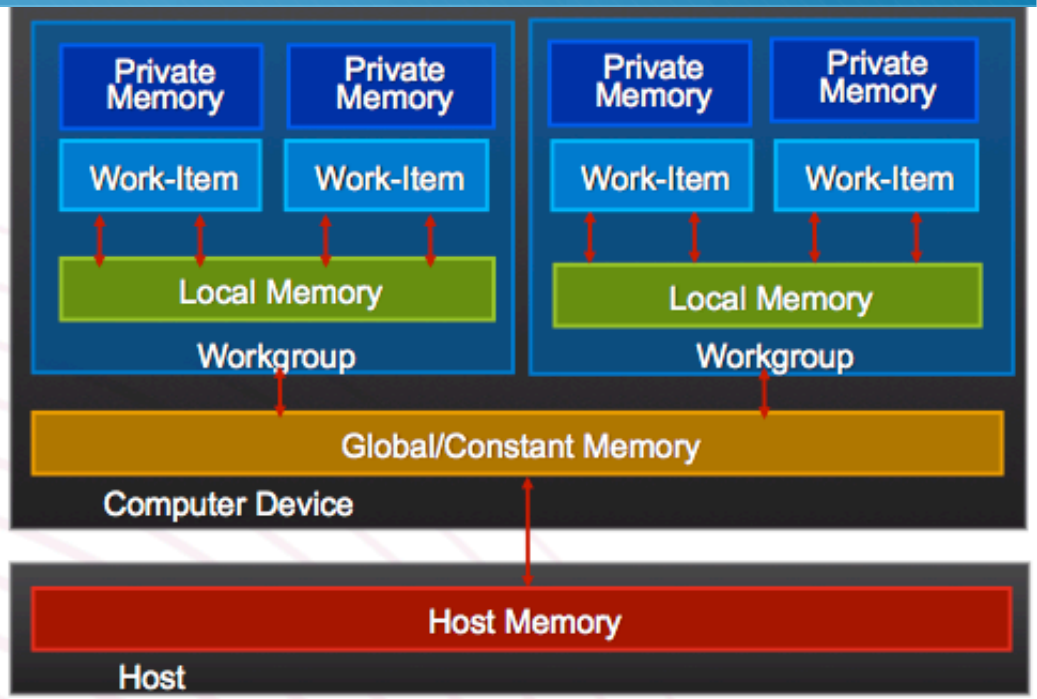
Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup

- Choose the dimensions that are "best" for your algorithm

OpenCL Memory Model

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Local Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU



- Memory management is explicit

You must move data from host -> global -> local *and* back

What makes OCL attractive?

- Allows explicit definition of parallel execution at multiple levels.
- It also opens the possibility of design space exploration of the execution platform's area/ performance ratio

OCL Features

- Implicit Independence between work-items and work groups
- Support for multiple disjoint address spaces
- No dynamic memory allocation
- Vector data types
- Recursion not supported.

2.Application Specific Processors (ASP)

- What is an ASP?
- Transport Triggered Architecture (TTA)
- TTA-based Co-design Environment (TCE)

What is ASP?

- Digital Signal Processors (DSP)
- Application Specific Set Processors(ASIP)
- Application Specific Integrated Circuit (ASIC)

Transport Triggered Architecture (TTA)

- Like VLIW architecture
  - Difference

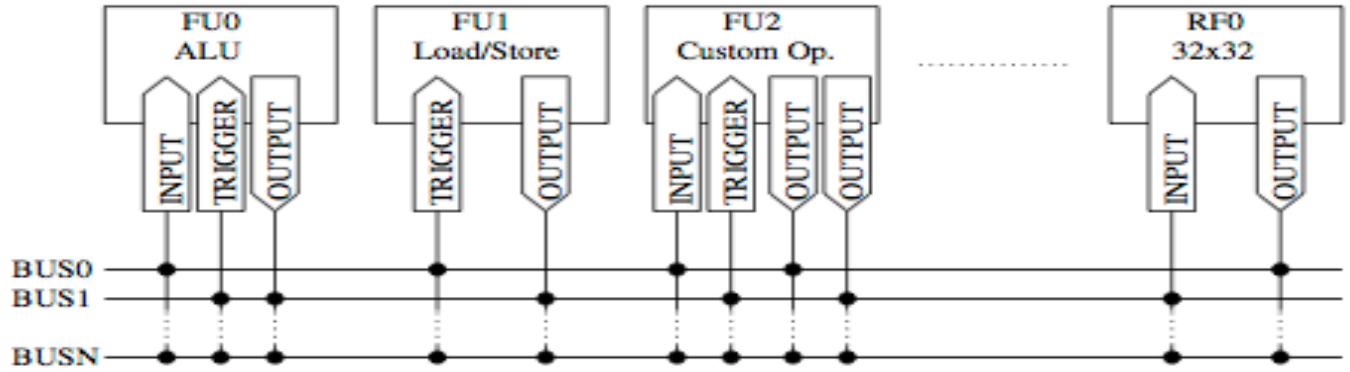


Fig. 1. Example of a TTA processor.

TTA (cont'd)

- Traditional Processor Architecture

```
add r3, r1, r2
```

- TTA

```
r1 -> ALU.operand1
r2 -> ALU.add.trigger
ALU.result -> r3
```

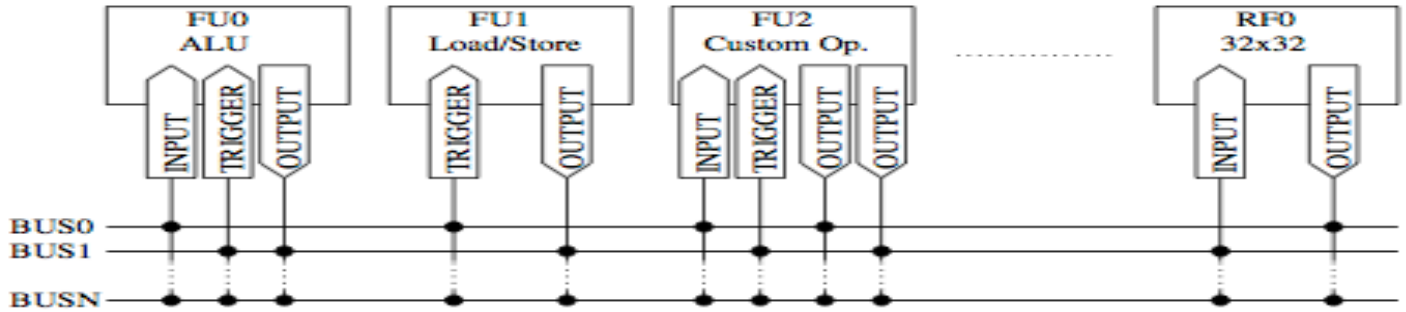


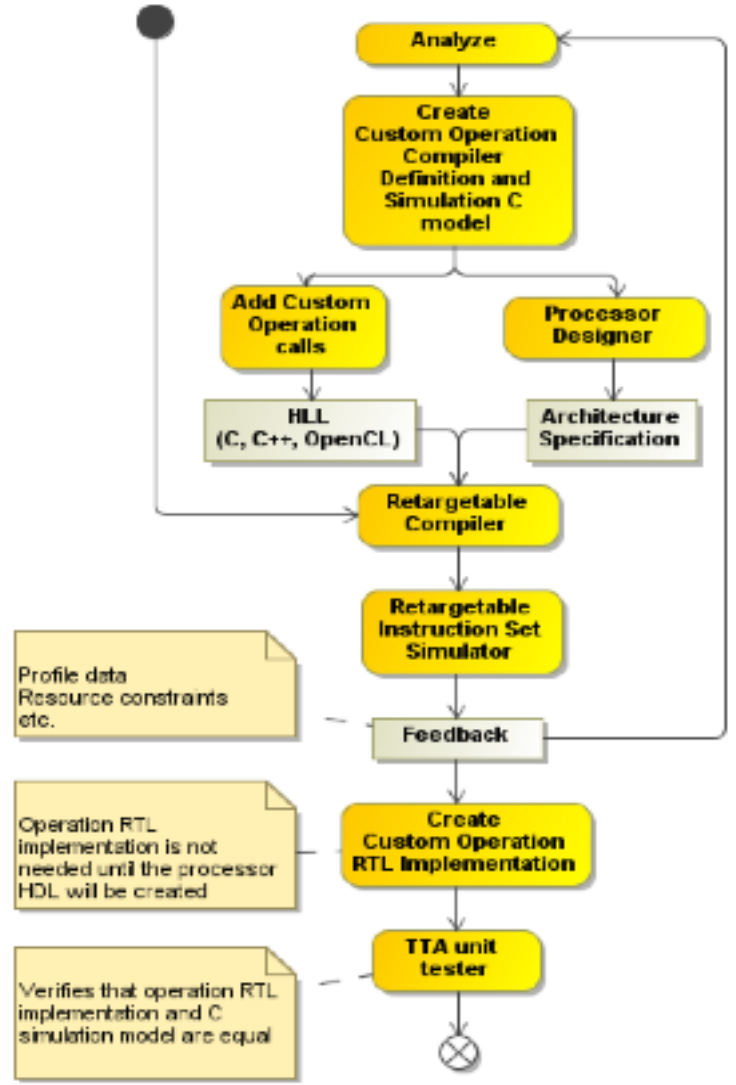
Fig. 1. Example of a TTA processor.

TTA-based Co-design Environment (TCE)

- Architecture description driven instruction set simulators
- A retarget able compiler
- A processor implementation generator supporting (Verilog, VHDL)

TCE(cont'd)

- Custom operations are added via OCL kernels
- TCE quality highly depends on the compiler.
  - Uses LLVM infrastructure



3.Practical Issues in compiling OCL

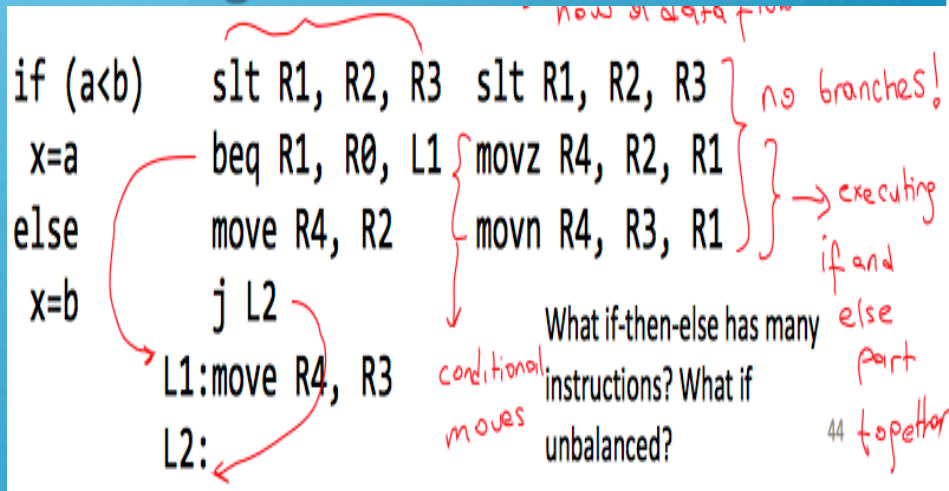
- Standalone Execution of OpenCL Applications
- Chaining Work-Items
- Work-item chaining Algorithm
- Efficient Instruction Scheduling of Work-Items
- Custom Operation Support

Standalone Execution of OpenCL Applications

- Proposed OCL to ASP design methodology starts from implementation and verification of the OCL application.
- OCL is designed to be portable across multiple platforms.
- 2 Paradigms
  - Standalone
  - Host/Device

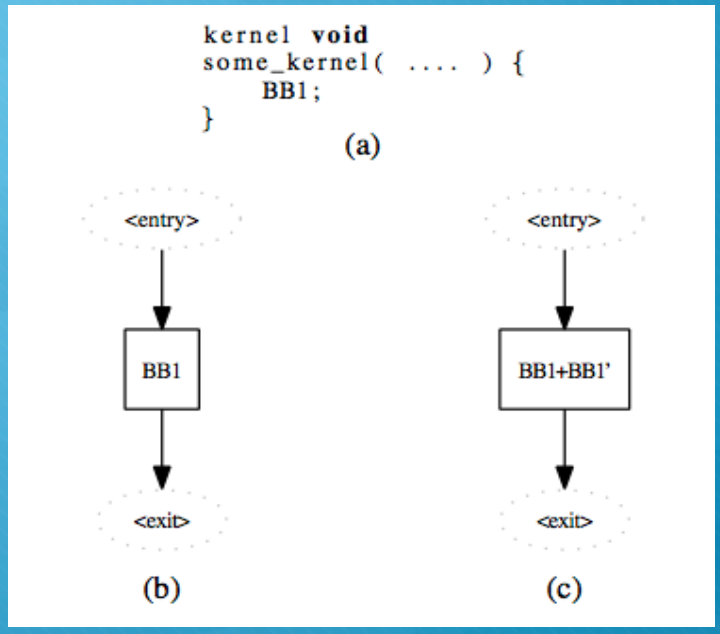
Standalone Execution of OCL Applications (cont'd)

- The paper choses the stand-alone
  - ASP act as the host, the compute device, and the compute unit.
  - FUs can be considered as processing elements
  - OCL memory model
    - Global & Constant Memory
    - Local & Private Memory
- Predicate aware scheduling
  - 2 executions to same FU



Chaining Work-Items

- Chain code from multiple work-items by just appending multiple instances of kernel code after each other
- Loop iterations in parallel
- Chaining

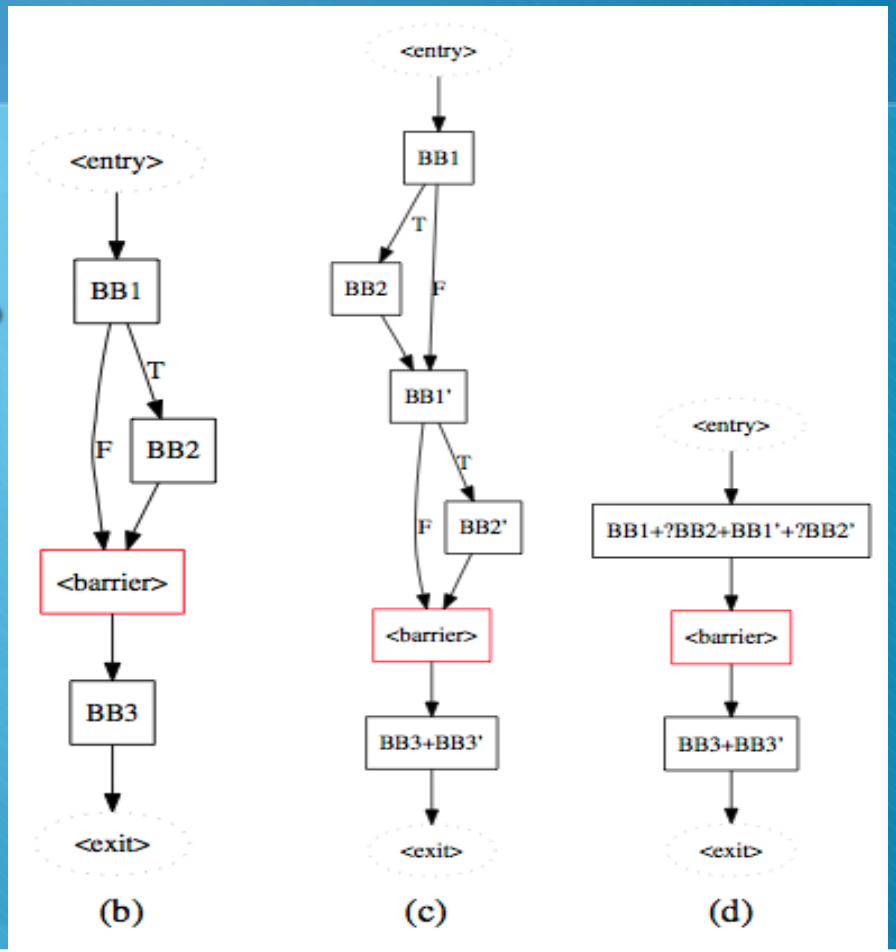


Chaining Work-Items (cont'd)

- Work-group barriers
- Barrier inside if/for stmt?

```
kernel void
some_kernel( .... ) {
    if (BB1) {
        BB2;
    }
    barrier();
    BB3;
}
```

(a)



Work-item Chaining Algorithm

- Flattening
- 2 Ways:
  - Creating code
  - Replicating code
- Parameterization

```
LLVMOpenCL(module)
1  for each Function f ∈ module
2       do if IsKernel(f)
3            then Flatten(f)
4                 DetectBarriers(f)
5                 for each Region r ∈ f
6                      do LoopRegion(r)
7                         ReplicateCode(r)
8  return module
```

Fig. 4. Work-item code replication algorithm.

Work-Item Chaining Alg (cont'd)

- Replication Algorithm
  - Basic loop unrolling
  - Mark instructions belonging to different work-items
- Chaining Algorithm
  - Easy-to-debug code, no optimizations
  - Several basic blocks connected by unconditional branches (to be combined into superblocks) – Like Trace Scheduling

Efficient Instruction Scheduling Of Work-Items

- TTAs are statically scheduled in compiler.
- Customized Register Allocator
  - Assign different registers for the chained work-items to allow them to be fully parallelized
  - Based in LLVM Linear Scan Register Allocator
  - Effective but sub-optimal

Efficient Instruction Scheduling (cont'd)

- Another parallelization inhibitor : memory accesses
  - Alias Analysis
- Assist to alias analysis:
  - All pointer arguments to the kernel function should not alias "restricted pointers"
  - Accesses to different spaces cannot alias. (Global and local)
  - Access through pointers to the constant memory can be assumed to be only reads.
  - Most importantly: in regions between work group barriers, the memory accesses of different work-items can be considered not to alias.

Custom Operation Support

- The use of custom operations
  - Special instructions , special function units
  - Software aided to fully hardware implementation
- TCE generates the headers and macros to use the custom operations
- OpenCL Extension API
  - Means to access to custom ops in the target processor.

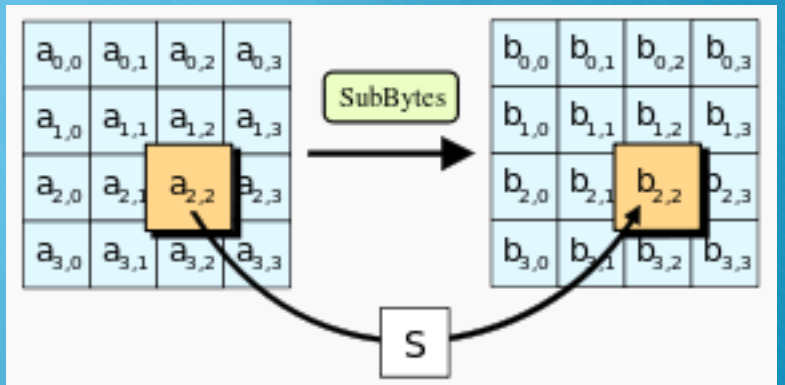
```
#ifdef cl_TCE_ADDSUB
 clADDSUBTCE(a, b, c, d);
#else
 c = a + b;
 d = a - b;
#endif
```

4.Experiments

- AES Encoder
- Algorithm
- Instruction Level Parallelism
- Custom Operations

AES Encoder

- 128 bits data and a key of 128 bits.
- Operations: substitutions, rotations and permutations using the 128 bits of data as 4x4 array of bytes.



The SubBytes step, one of four stages in a round of AES

Algorithm

- Key Expansion
  - Take 128 bit key and generate 1408-bit expanded key
  - Needs to be done if the key doesn't change
- Encryption / Decryption
  - Done on each block of 128 bits on source data
- Implemented as an OpenCL Kernel

Algorithm (cont'd)

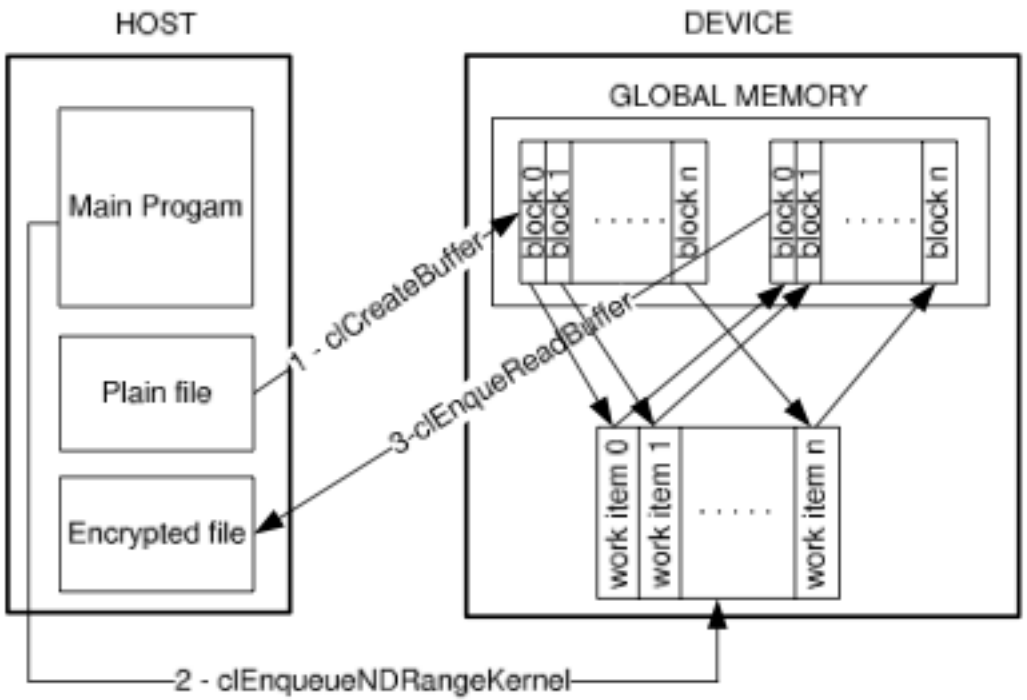


Fig. 6. The OpenCL AES encryption implementation.

Instruction-level Parallelism

- Experiment 1 : Verifying the ILP scalability of the OpenCL implementation.
  - Setup: architecture provided enough resources for the program to be limited only by its data dependencies.
  - OpenCL application was compiled for this architecture with one, two and 4 parallel work items

Parallel WIs	cycles	speedup
1	35,729	1.00
2	18,209	1.96
4	9,505	3.76

Custom Operations

- Experiment 2: use of custom hardware operations to accelerate the application using the OpenCL C extension API
- 2 Custom Operations:
  - MUL_GAL
  - SUBSHIFT

resource	multiplicity	notes
Arithmetic-logic unit	3	1 cycle latency
Register file	3	16 registers per file
Load/Store unit	1	2 cycle load latency
32-bit multiplier unit	1	3 cycle latency

- Xilinx Virtex 5 => 191 MHZ
- Stratix II => 149 MHZ

Custom Operations (cont'd)

- MUL_GAL, a multiplication of 2 integers in the Galois field GF(2^8)
- SUBSHIFT, involves searching in a look-up table, substituting and mixing some elements of an 4x4 array.

architecture	cycles	speedup	KB/s at 100 MHz
AESTTA	1,119,415		366
AESTTA+MUL_GAL	450,490	2.5	909
AESTTA+MUL_GAL+SS	286,778	3.9	1,428

TABLE III
SPEEDUPS FROM CUSTOM OPERATIONS.

5.Conclusion

- the parallel application description capabilities of OpenCL make it possible to scale the single core performance of the ASP design efficiently by increasing the number of parallel work-items and datapath resources.
- The custom operation support was verified by using two non-trivial custom operations to accelerate the AES ASP design.

Any Questions?

Loop Unrolling

```
for (i=0; i<N; i++)
    B[i] = A[i] + C;
```

Unroll inner loop to perform 4 iterations at once

```
for (i=0; i<N; i+=4)
{
    B[i]   = A[i] + C;
    B[i+1] = A[i+1] + C;
    B[i+2] = A[i+2] + C;
    B[i+3] = A[i+3] + C;
}
```

Software Pipelining (I)

Unroll 4 ways

```
loop: lw F1, 0(r1)
      lw F2, 4(r1)
      lw F3, 8(r1)
      lw F4, 12(r1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```

Schedule

	Int1	Int 2	M1	M2	FP+	FPx	
loop:			lw F1				1
			lw F2				2
			lw F3				3
	add R1		lw F4		add.s F5		4
					add.s F6		5
					add.s F7		6
					add.s F8		7
			sw F5				8
			sw F6				9
			sw F7				10
	add R2	bne	sw F8				11

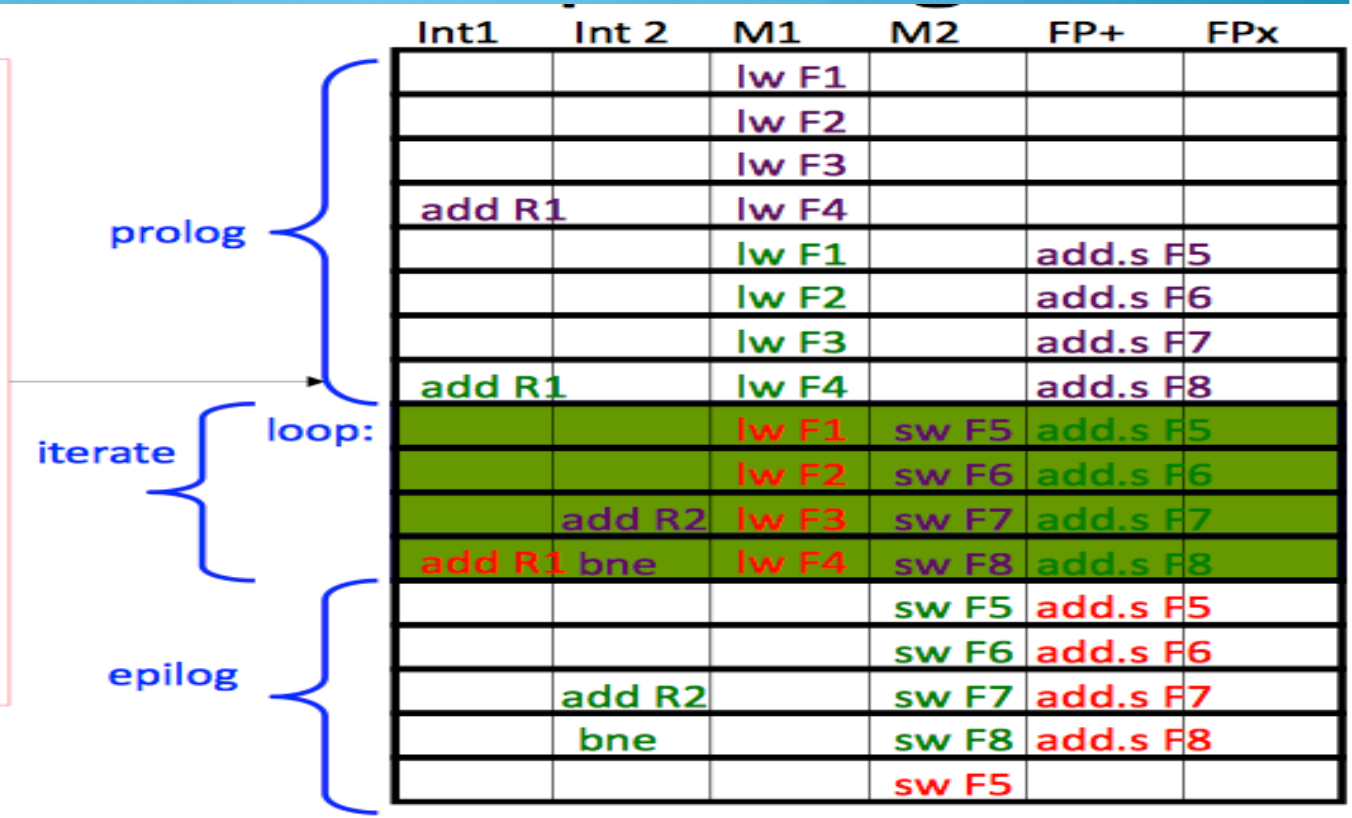
How many FLOPS/cycle?

4 add.s / 11 cycles = 0.36

Software Pipelining (II)

Unroll 4 ways first

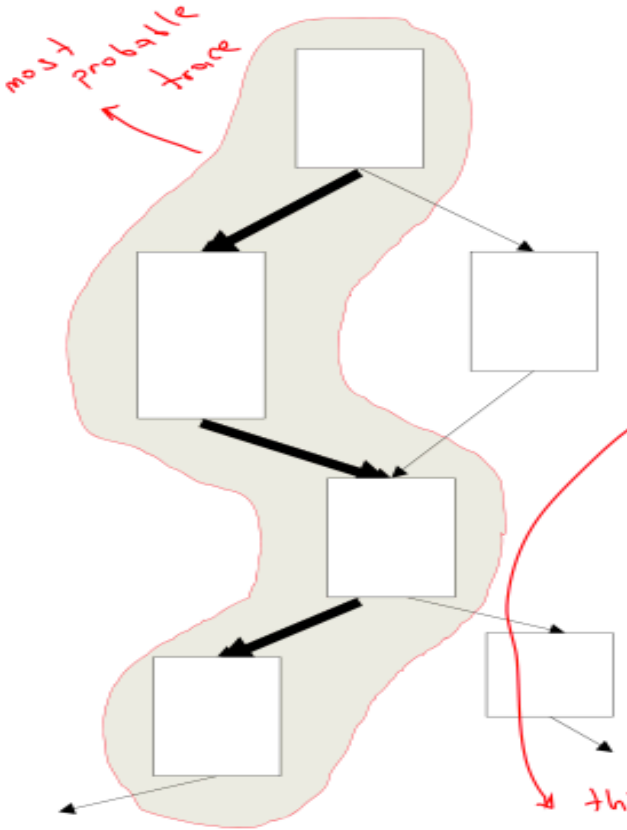
```
loop: lw F1, 0(R1)
      lw F2, 4(R1)
      lw F3, 8(R1)
      lw F4, 12(R1)
      addiu R1, R1, 16
      add.s F5, F0, F1
      add.s F6, F0, F2
      add.s F7, F0, F3
      add.s F8, F0, F4
      sw F5, 0(R2)
      sw F6, 4(R2)
      sw F7, 8(R2)
      sw F8, 12(R2)
      addiu R2, R2, 16
      bne R1, R3, loop
```



Anti-Dependency

- B = 3 (I)
- A = B + 1 (II)
- B = 7 (III)
- WAR (Write After Read)
  - The ordering of 2 and 3 can't be changed nor they can be executed in parallel.

Trace Scheduling



- Pick string of basic blocks, a *trace*, that represents most frequent branch path
- Use profiling feedback or compiler heuristics to find common branch paths
- Schedule whole "trace" at once
- Add fixup code to cope with branches jumping out of trace

- this doesn't mean you can't go out of this trace, if you do, you need recovery code.

this trace is basically creates a monolithic code