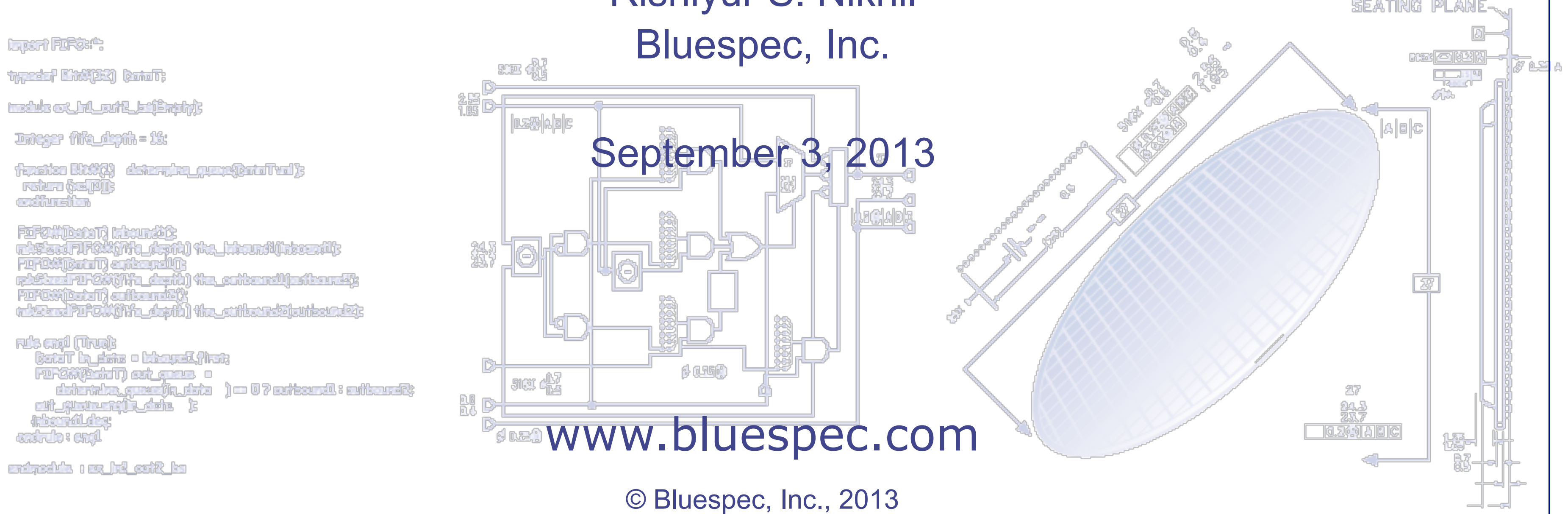
Notes on Blocked Dense Matrix Multiply example written in BSV using BClib for Convey Platforms

Rishiyur S. Nikhil
Bluespec, Inc.

September 3, 2013



www.bluespec.com

Introduction

- This document provides some explanatory notes on an example “Blocked Dense Matrix Multiply” code for Convey platforms, written in the BSV language and using Bluespec’s BClib library for Convey platforms.

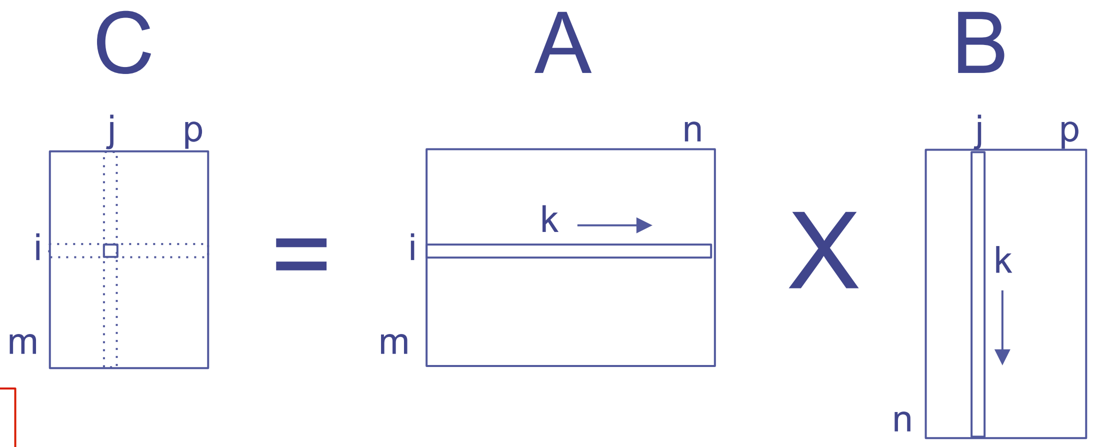
- The purpose of this example is to illustrate how to write codes with complex memory access patterns using BSV/BClib. Optimization of the core arithmetic is a separate problem, and has not yet been addressed here.
    - Currently it works on matrices of 64-bit signed integers, and the core BlockMAC is quite serial. It would be easy to substitute other 64-bit arithmetic such as floating point, fixed-point, complex, or other). The BlockMAC also needs to be parallelized for better performance. See additional notes at end of this document.

- This example is very flexible with respect to memory layout of the input and output matrices, memory layout of blocks within these matrices, etc.

Matrix multiplication basics

C = A x B

Each element C_{ij} =
inner_product (row A_i,
column B_j)



```
for (i = 0..m-1)
    for (j = 0..p-1)
        C[i,j] = 0;
        for (k = 0..n-1)
            C[i,j] = C[i,j] + A[i,k] x B[k,j]
```

"inner product"

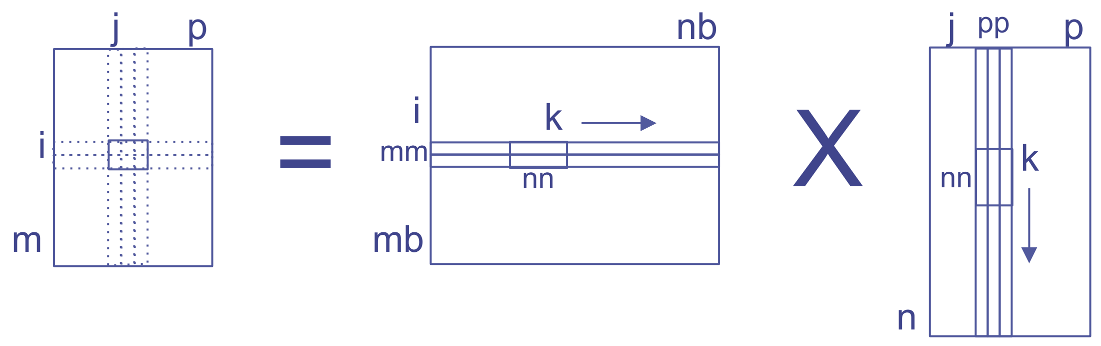
"multiply-accumulate" a.k.a. "MAC"

In this doc, and in the BSV code, we strictly follow certain variable-naming conventions, as shown in the diagram above:

- m, n, p for dimensions of A, B and C
- I, j, k for indexes along the m, p and k dimensions, respectively

bluespec

"Blocked" Matrix multiplication ("BMM")



Instead of iterating over individual rows and columns of scalars, we iterate over "blocks" of rows and columns.

- mm, nn, pp: dimensions of blocks in A, B and C (see figure above)

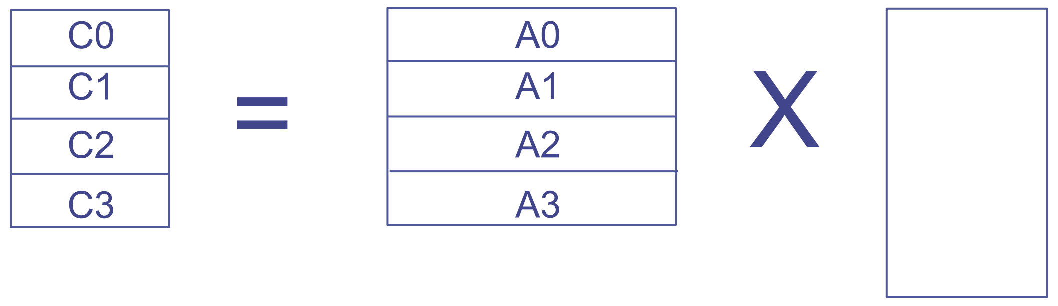
Each inner-product, therefore, involves

- "multiplication" of (mm x nn) blocks from A with (nn x pp) blocks from B
- "addition" of "(mm x nn)" blocks (blocks of C)

Here "multiplication" and "addition" are, simply, matrix multiplication of scalars and matrix addition of scalars (unblocked).

Why? If a processor reads entire blocks into "local" memory and performs block multiplications and additions locally, the total number of main memory references can be reduced (by a factor of the block size), which can improve overall performance.

BMM for Convey: top-level parallelization



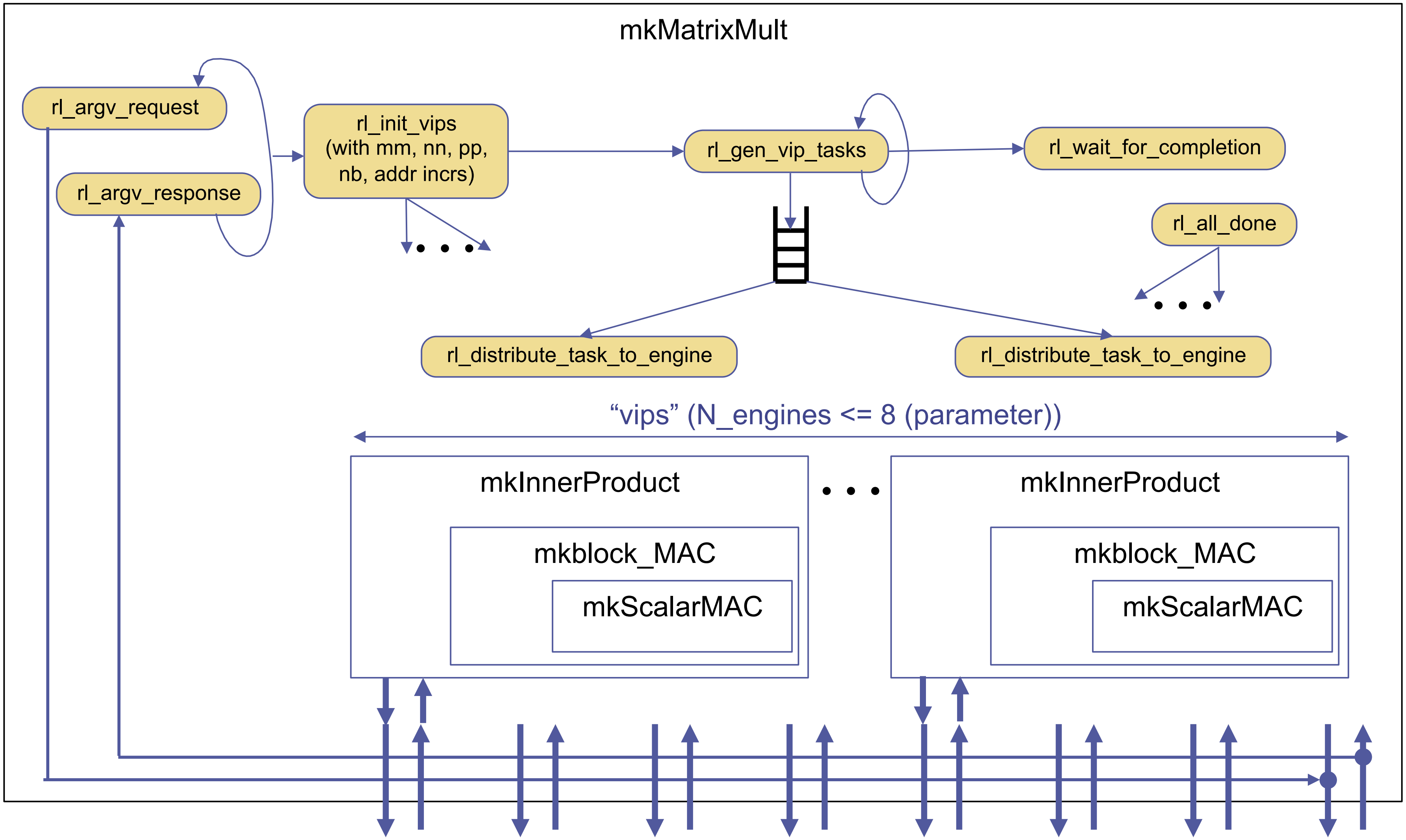
For the top level of parallelization, one option is to simply slice the A and C matrices horizontally into four parts, so that:

• FPGA 0 does:	C0 = A0 x B
• FPGA 1 does:	C1 = A1 x B
• FPGA 2 does:	C2 = A2 x B
• FPGA 3 does:	C3 = A3 x B

Each of these is just an independent matrix multiplication, so all FPGAs can have the same hardware setup; we just provide them with different matrix addresses on startup.

This code will equally well support other parallelizations, such as slicing B and C vertically into four parts (see later slide on “input arguments”).

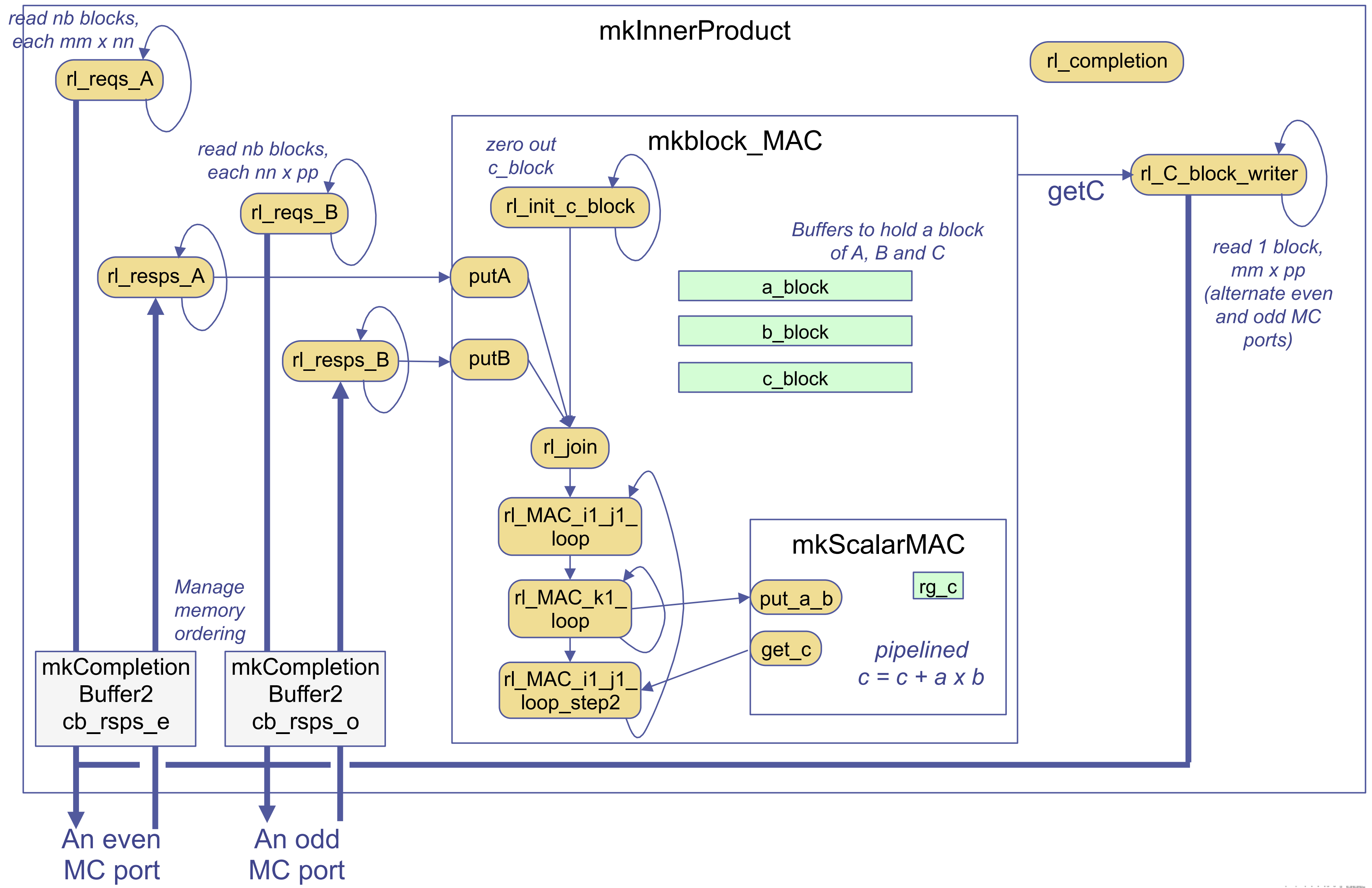
BMM for Convey: each FPGA



Each mkInnerProduct engine is attached to one Convey MC even/odd pair. The argv request/response rules use the last (MC 7) even/odd pair briefly, at the start of the computation.

8 MCs (Convey memory ports)

BMM for Convey: each InnerProduct engine

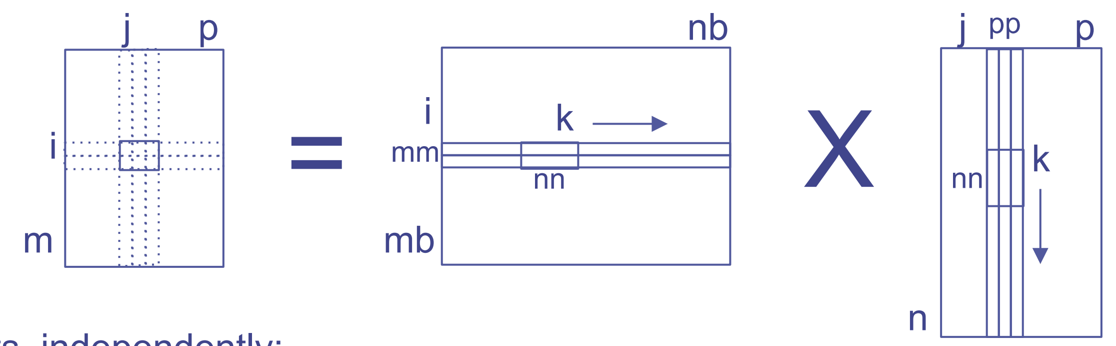


bluespec

BMM for Convey: memory layout independence

The code can be used with many possible memory layouts for the matrices, because it takes arguments specifying the following (we do not assume square matrices or blocks):

argument	**Offset in arg block**	**comment**
pA, pB, pC	[1], [2], [3]	Address of A[0,0], B[0,0], C[0,0]
mb, nb, pb	[4], [5], [6]	# of blocks in m, n and p dimensions
mm, nn, pp	[7], [8], [9]	Block sizes A:mmxnn, B:nnxpp, C:mmxpp
dAi1, dAk1, dAib, dAkb	[10], [11], [12], [13]	Address increment for A, by 1 row, 1 col, by mb rows, by nb cols
dBk1, dBj1, dBkb, dBjB	[14], [15], [16], [17]	Address increment for B, by 1 row, 1 col, by nb rows, by pb cols
dCi1, dCj1, dCib, dCjb	[18], [19], [20], [21]	Address increment for C, by 1 row, 1 col, by mb rows, by pb cols



Note: this supports, independently:

- Row-major or column-major ordering of blocks
- Row-major or column-major ordering of elements within blocks
- Padding between elements, padding between blocks
- Different layouts for A, B, and C
- Non-square, non-power-of-2-size matrices and blocks
- etc.

BMM for Convey: input arguments

The host software prepares the matrices, and passes these arguments to the FPGAs in an arg block, which is an array of 1024 64-bit words, 256 per FPGA.

The arguments can in principle be completely different for each FPGA. If we divide work across FPGAs by horizontally slicing A and C, only pA and pC will be different for the different FPGAs. If we divide work by vertically slicing B and C, only pB and pC will be different.

We only have 22 args for each FPGA, so a 256-word array for each FPGA is overkill, but this part of the code can be reused as-is for different apps with different # of args.

The single argument to the top-level mkApp_HW module in all four FPGAs is a pointer to this block.

This module, in turn, calls its mkMatrixMult.start() method with the pointer + (FPGA_id << 11), which points at the 256-word block for current FPGA.

BMM for Convey: notes

Synthesis notes:

- When doing a ‘make’ to compile the bitfile, in Personality/Makefile.include,
  - set MC_XBAR=1
  - set MC_READ_ORDER=1
- Before doing ‘make’ to create the bitfile, first do ‘make cae_fpga.xst’, and edit the cae_fpga.xst file to change the following Xilinx options:
  - -fsm_extract YES => NO
  - -resource_sharing YES => NO
  - -equivalent_register_removal YES => NO

Synthesis results (for Convey HC-1ex)

- N_engines = 8 (per FPGA); signed 64-bit integer arithmetic
- Resource utilization: LUTs 23%, Registers 14%, DSPs 9%, Block RAMS: 4%
  - So: there is plenty of space for optimization of the core arithmetic, BlockMAC and ScalarMAC
- Zero timing errors (no negative slack)

To convert to floating-point arithmetic, simply substitute the mkScalarMAC block

The current mkScalarMAC block has a naïve integer multiplication, with too many stages. Instead, this should just use a wrapped version of an existing optimized core, from Xilinx or elsewhere.

Currently, the inner conventional matrix-multiply of two blocks, inside mkBlock_MAC, is sequential. This should be parallelized (e.g., using a systolic array).