X3PU

General Purpose Parallel Processing Unit used as Accelerator in a Heterogenous Computing System

X3PU team

Executive summary

General Purpose Parallel Processing Unit (X3PU) that we propose as accelerator is thought of as part of a heterogeneous computing system where it is necessary to accelerate critical functions (for **AI, automotive, blockchain, protein folding,** *...) in terms of* ***execution time*** *and* ***energy consumption****. Equally important is the* ***area*** *occupied on silicon (because the manufacturing price increases super-linearly with the area) if the aim is to create an integrated circuit. No less important is the* ***programmability*** *that is realized at the user's level in the currently used languages (C, C++, Python) that call on function libraries implemented in hardware by our accelerator (off-the-shelf solutions use complex environments; for example CUDA in the case of nVIDIA) .*

Our proposal is compared with current solutions on the market and results, for functions frequently used in current applications, in an approximately 11-fold reduction in energy and 3-fold reduction in area for the same amount of computation.

The technology we have developed can be exploited in at least three ways: (1) implementation in FPGA, (2) implementation in silicon and (3) delivery of IPs.

The solution was implemented in a Silicon Valley start-up. The last version, in 2008, was an application in the video field (1024 processing elements, in 65 nm technology, to speed up the frame rate conversion operation).

The current solution is a much improved one, implemented in FPGA. The initial financing is necessary to develop the Software Development Kit (SDK) and the kernel of some function libraries from the category of those currently used. The SDK is necessary to convince users to use the technology and a meaningful kernel will allow understanding the advantages offered.

Our *X3PU* is the accelerator component of a heterogenous computing system (see Figure 1. It is designed to execute intense parts of an application (the most time, energy and area consuming) while the complex part is executed by the HOST computer part of the system. *X3PU* is a ***configurable & parameterizable*** many-cell computational system.

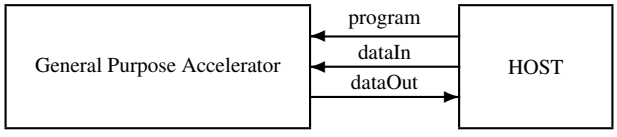

Figure 1: Heterogenous system.

Why such a solution? Many of the currently used applications require particularly intensive calculations in the sense of the simplicity of the description accompanied by a very large number of operations, which leads to prohibitive execution times and energy. A solution is required that accelerates the computation and minimizes the energy consumed. The solution proposed by the scientific community and accepted by the corporate space is heterogeneous computing.

Where does our solution come in? We propose a general-purpose accelerator ($X3PU$) because the main weakness of the current parallel solution for intense computations are due to:

- the use of ad-hoc hardware structures designed more from geometrical considerations than from the consideration of computational aspects,
- using specific accelerators to solve general-purpose computational issues (the oxymoronic example of GPGPU is typical).

The structural and architectural inadequacy of current solutions leads to high consumption of energy and silicon area (the manufacturing price of a chip increases super-linearly with the area).

$X3PU$ **programming** is carried out at several levels as follows:

- level 1: in assembly language at the accelerator level for the development of function libraries using data structures limited by the hardware structure, called elementary libraries,
- level 2: in a high-level language, using elementary libraries to develop libraries of dedicated or user-defined functions,
- level 3: in a high-level language, using the libraries developed on level 2, for developing applications.

In Appendix 1 is shown a subset of functions developed at the level 1. The functions a used for writing program for matrix-matrix multiplication at the level 2 (see Appendix 2). Programming level 1 is accessible to internal developers, while levels 2 and 3 are also accessible to users.

The software components built on top of $X3PU$ are represented in Figure 2[1]. While the flow view of the software architecture is represented in Figure 3[2].

[1] SDK: Software Development Kit; IDE: Integrated Development Environment; XRT: $X3PU$ Run-Time environment.

[2] ONNX: Open Neural Network Exchange, is an open format built to represent machine learning models. ONNX defines a common set of operators - the building blocks of machine learning and deep learning models - and a common file format to enable AI developers to use models with a variety of frameworks, tools, runtimes, and compilers.

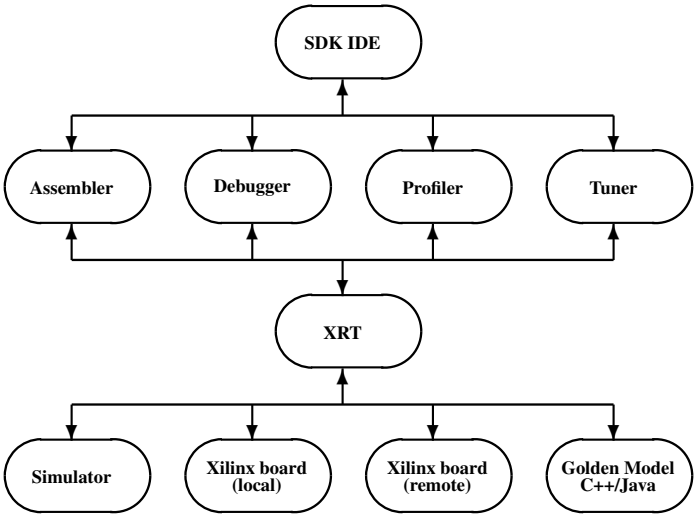

Figure 2: Software architecture: component view

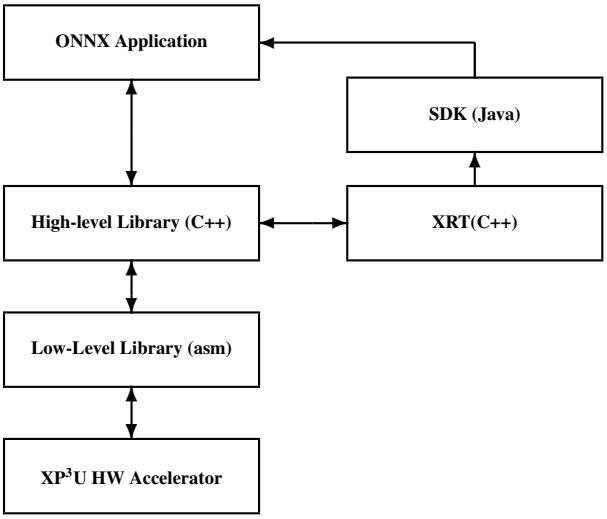

Figure 3: Software architecture: flow view

*X*3*PU* **Structure** (see Figure 4) consists of:

- MAP: a linear array of p cells, each with its data memory and an execution unit
- CONTROLLER: which sends an instruction in each clock cycle to be executed in the active cells of the MAP
- a distribution pipeline network of logarithmic depth

- a REDUCE network of logarithmic depth that takes a vector from the MAP and sends a scalar to the CONTROLLER (sum, min, max, ...)
- a logarithmic depth SCAN network that takes a vector from the MAP and sends a vector back to the MAP (permutation, prefix, ...)

The CONTROL receives the program from the HOST and exchanges data with the HOST's memory.

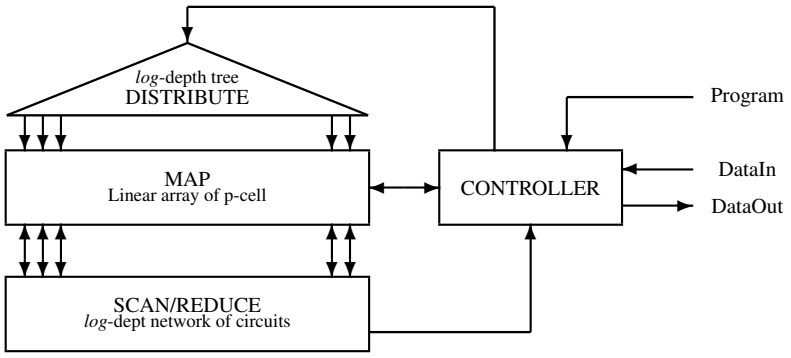

Figure 4: *X3PU*.

The entire structure is parameterizable (the size of the word with which the MAP execution units operate, the size of the local memories in the MAP cell, the number of cells in the MAP) and configurable (the SCAN and REDUCE functions, floating point operation, ...). Our solution does not require special technologies to achieve the performance we claim. The advantages of our solution come from organizational and architectural improvements only.

History: several versions of *X3PU* have been implemented in silicon in the first decade of the century in a Silicon Valley startup (see `http://users.dcae.pub.ro/~gstefan/2ndLevel/connex.html`). The 90 nm version is represented in Figure 5. Last, a 65nm version was used to implement the frame rate conversion function for dual HD video stream. At the current stage we have an FPGA implementation that is programmed in assembly language (working prototype, on PYNQ-Z2 development board, for p = 128).

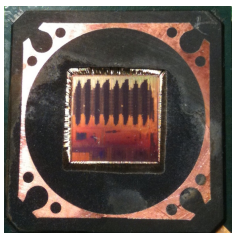

Figure 5: The 90 nm version of CA1024, a many-core of 1024-cell *X3PU* (16-bit execution unit, 0.5KB local memory).

August 20–22, 2006: the BA1024 chip was presented in Memorial Auditorium of Stanford University in *Hot Chips: A Symposium on High Performance Chips*.

July 2007: the first real & complex application (HDTV postprocessing for frame-rate conversion) running on BA1024 was presented in Department of Devices, Circuits and Architectures of Politehnica University of Bucharest (the software team coordinated by Bogdan Mitu, Marius Stoian, and Radu Weiss)

Performances & comparisons of our proposal were made by investigating the calculation of computationally intensive functions involved in current applications such as: AI, automotive, bio-computing, DSP, etc. For some critical functions (for example matrix-matrix multiplication), compared with Nvidia GA100 chip, implemented in 7nm, we have evaluated (see Appendix E) the following improvements:

- $(11 \pm 20\%)\times$ less energy
- $(3 \pm 10\%)\times$ less area (which translates in $> 3\times$ production price due to the decreasing yield with chip area)

The architectural acceleration, compared to single core machine of similar characteristics, are supra-linear:

- matrix-matrix multiplication: $\sim$6p, obtained by running the program on our system (see Appendix C written using the library partially described in Appendix B whose development is exemplified in Appendix A) and compared with one written for an x86 mono-core system (the assembly code resulting from compilation is in Appendix D)
- FFT: $\sim$1.5p
- pooling: $\sim$2.1p

(p being the number of processing cores used).

Stage & future work is summarized in Figure 6.

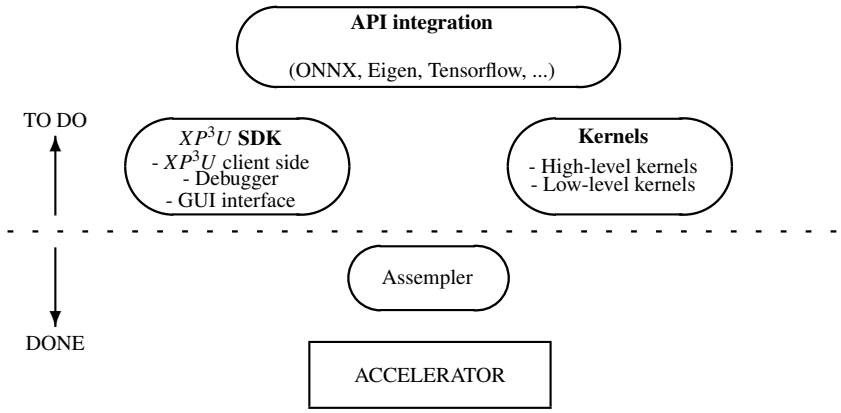

Figure 6:

References

[1] Mihaela Malița, Gheorghe Ștefan, Dominique Thiébaut (2007) Not Multi-, but Many-Core: Designing Integral Parallel Architectures for Embedded Computation, *ACM SIGARCH Computer Architecture News*, **35**(5)32:38, Special issue: ALPS '07 - Advanced low power systems; communication at International Workshop on Advanced Low Power Systems held in conjunction with 21st International Conference on Supercomputing June 17, 2007 Seattle, WA, USA.

[2] Gheorghe Ştefan (2006) The CA1024: A Massively Parallel Processor for Cost-Effective HDTV, *SPRING PROCESSOR FORUM JAPAN*, June 8-9, 2006, Tokyo.

[3] Gheorghe Ştefan, Anand Sheel, Bogdan Mîţu, Tom Thomson, Dan Tomescu (2006) The CA1024: A Fully Programmable System-On-Chip for Cost-Effective HDTV Media Processing, *Hot Chips: A Symposium on High Performance Chips*, Memorial Auditorium, Stanford University, August 20 to 22, 2006.

[4] Gheorghe M. Ștefan, Mihaela Malița (2014) Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation, *18th International Conference on Circuits, Systems, Communications and Computers* , Santorini Island, Greece, pp 582-597. http://www.inase.org/library/2014/santorini/bypaper/COMPUTERS/COMPUTERS2-42.pdf

[5] User's Guide — NVIDIA Docs (2023) Matrix Multiplication Background, https://docs.nvidia.com/deeplearning/performance/pdf/Matrix-Multiplication-Background-User-Guide.pdf

[6] NVIDIA GA100 (2023) NVIDIA GA100, https://www.techpowerup.com/$X3PU$-specs/nvidia-ga100.g931

APPENDICES

A Assembly code for dense matrix multiplication on $X3PU$

The assembly code for multiplying square matrices is listed below. It includes the transfer of the two matrices from the host processor and of the transfer back of the result into the host's memory. The execution time is given by the following relation (extracted from the code and verified by running the program in FPGA environment):

$$T_{Multiply} + T_{Transfer} = ((2p^2 + plog_2 p + 9p + 5) + (1.5p^2 + 17p + 6))clockCycles$$

```
;*******************************************************************************
ASSEMBLY PROGRAM for p x p dense matrix multiplication & data transfer on p-cell GP^3A
;*******************************************************************************
;LOAD M1 and M2
            vload 31        vload 15
            store 0         addrld
    label0  vload 8         nop
            getv            nop
            load 0          ioload
            brzdec label1   ristore 1
            store 0         nop
            jmp label0      nop

;MULTIPLICATION  n=16  end of M1=31
    label1  vload 15        vload 31
            store 0         addrld
            vload 47        nop
            store 3         nop
            vload 32        nop
            nop             nop
            vadd 1          caload
            store 2         store 0
            vload 16        rload

  ;THE MAIN LOOP accessed n^2 times
    label2  redins          mult 0
            brnzdec label2  riload -1
  ;END OF THE MAIN LOOP

            load 3          nop
            vadd 1          nop
            store 3         srload
            load 2          cstore
            nop             nop
            vadd 1          caload
            store 2         store 0
            load 0          vload 31
            brzdec label3   addrld
            store 0         nop
            vload 16        rload
            jmp label2      nop

;STORE THE RESULT
    label3  vload 16        vload 47
            store 5         addrld
            nop             nop
```

```
label4  nop             riload 1
        load 5          iostore
        brzdec label5   nop
        store 5         nop
        nop             nop
        vload 7         nop
        sendv           nop
        jmp label4      nop
label5  setint          nop
```

B Subset of the library for linear algebra

A subset of functions belonging to the library of functions `theKernel16.v` is listed below. These functions are used by the host computer as part of a hardware implemented library of functions. All these functions are implemented in assembler (see in Appendix A an example of assembly code).

START : start cycles counter

STOP : stop cycles counter

INTRQ : send interrupt and cycles counter

MM_EWO(destination, source1, source2, linesNr, operation, waitMatricesNo) : element-wise operation on matrix

SM_MULT(destination, scalar, source, linesNr, waitMatricesNo) : scalar-matrix multiplication

MM_MULT(destination, source1, source2, linesNr, waitMatricesNo) : matrix-matrix multiplication

MM_MAC(destination, source1, source2, linesNr, waitMatricesNo) : matrix-matrix multiplication & accumulate

SEND_MATRIX_ARRAY(addr, nr_lines, nr_columns) : Data Transfer Engine command that performs the transfer of a data matrix in the data memory of the Array. This command needs to be followed by three parameters: the internal data memory address where the store will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, each data line will be padded with zeros.

GET_MATRIX_ARRAY(addr, nr_lines, nr_columns, wait_result) : Data Transfer Engine command that performs the transfer of a data matrix from the data memory of the Array to the Data Output FIFO. This command must be followed by three parameters: the internal data memory address where the read will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the array, the transfer of a data line will be considered finished after shifting out only the needed data. If wait_result is 1, the transfer will wait for a result to be marked as ready by the Controller.

SEND_MATRIX_CTRL(addr, nr_lines, nr_columns) : Data Transfer Engine command that performs the transfer of a data matrix in the data memory of the Controller. This command needs to be followed by three parameters: the internal data memory address where the store will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the Array, each data line will be padded with zeros.

GET_MATRIX_CTRL(addr, nr_lines, nr_columns, wait_result) : Data Transfer Engine command that performs the transfer of a data matrix from the data memory of the Controller to the Data Output FIFO. This command must be followed by three parameters: the internal data memory address where the read will start, the number of lines, and the number of columns. If the number of columns is smaller than the number of cells in the array, the transfer of a data line will be considered finished after shifting out only the needed data. wait_result is 1, the transfer will wait for a result to be marked as ready by the Controller.

cWAITMATW(scalar) : Wait for Matrices to be Written: Instruction for the Controller that tells it to perform no operation until the Data Transfer Engine confirms that a number of scalar matrices needed for the processing sequence are transferred into the Array's internal data memory.

cRESREADY : Result Ready: Instruction for the Controller to acknowledge the Data Transfer Engine that a processing sequence is finished and the result is ready to be read.

C Code on HOST

Using mainly functions exemplified in Appendix B, the program for dense matrix multiplication defined for large matrices is listed below. The size of matrices is MAT_DIM, a multiple of p, the number of cells of our accelerator.

```
/*******************************************************************************
THE PROGRAM: matrix multiplication
*******************************************************************************/
// THE PROGRAM USED TO INITIALIZE THE ACCELERATOR
            pload   activate
            nop     nop
            nop     rednop
            `include "theKernel16.v"
            prun 0
// THE EXECUTION
 START;                             // start the cycle counter
 for(index2 = 0; index2 < (MAT_DIM/NR_CELLS)**2; index2 = index2 + 1) begin
     SEND_MATRIX(16, 16, 16);       // write dest=16,  #Lines=16, #Columns=16
     SEND_MATRIX(144, 16, 16);      // write dest=144, #Lines=16, #Columns=16
     SEND_MATRIX(512, 16, 16);      // write dest=512, #Lines=16, #Columns=16
     SEND_MATRIX(640, 16, 16);      // write dest=640, #Lines=16, #Columns=16
     MM_MULT(272, 16, 144, 16, 2); // dest=272, left=16, right=144, #Lines=16, #Mat=2
     MM_MAC(272, 512, 640, 16, 2); // dest=272, left=512, right=640, #Lines=16, #Mat=2
     if(MAT_DIM/NR_CELLS - 1 > 1) begin
         for(index = 0; index < (MAT_DIM/NR_CELLS)/2 - 1; index = index + 1) begin
             WAIT_RESULT_READY();          // wait for result to be ready
             SEND_MATRIX(16, 16, 16);      // write dest=16,  #Lines=16, #Columns=16
             SEND_MATRIX(144, 16, 16);     // write dest=144, #Lines=16, #Columns=16
             WAIT_RESULT_READY();          // wait for result to be ready
             SEND_MATRIX(512, 16, 16);     // write dest=512, #Lines=16, #Columns=16
             SEND_MATRIX(640, 16, 16);     // write dest=640, #Lines=16, #Columns=16
             MM_MAC(272, 16, 144, 16, 2);  // dest=272, left=16,  right=144, #Lines=16,
                                           // #Mat=2
             MM_MAC(272, 512, 640, 16, 2); // dest=272, left=512, right=640, #Lines=16,
                                           // #Mat=2
         end
     end
     WAIT_RESULT_READY();
     WAIT_RESULT_READY();
     GET_MATRIX(272, 16, 16); // read from dest=272, #Lines=16, #Columns=16
     INTRQ;                   // send the interrupt
 end
 STOP;                        // stop the cycle counter
```

D Assembly code for dense matrix multiplication on mono-core x86 system

Compiling a program to multiply two matrices of 16×16 elements generated the executable code below. The code also includes the generation of the two matrices. The main loop, which repeats 16^3 times, is labeled with .L13. We will consider, covering, the fact that the mono-core processor executes on average more than 1.22 instructions per clock cycle (the execution report provides for the entire program 1.22 instructions per cycle). In this case, the execution time considered for matrix multiplication is $22 \times N^3$.

$$T_{multiply+Transfer} \simeq 22N^3 clockCycles$$

```
 .file  "matrMul.c"
    .text
    .globl  main
    .type   main, @function
main:
.LFB5:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $12320, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $0, -12316(%rbp)
    jmp .L2
.L5:
    movl    $0, -12312(%rbp)
    jmp .L3
.L4:
    call    random@PLT
    movl    %eax, %ecx
    movl    -12312(%rbp), %eax
    cltq
    movl    -12316(%rbp), %edx
    movslq  %edx, %rdx
    salq    $5, %rdx
    addq    %rdx, %rax
    movl    %ecx, -12304(%rbp,%rax,4)
    addl    $1, -12312(%rbp)
.L3:
    cmpl    $31, -12312(%rbp)
    jle .L4
    addl    $1, -12316(%rbp)
.L2:
    cmpl    $31, -12316(%rbp)
    jle .L5
    movl    $0, -12316(%rbp)
    jmp .L6
.L9:
    movl    $0, -12312(%rbp)
    jmp .L7
.L8:
    call    random@PLT
    movl    %eax, %ecx
```

```
    movl    -12312(%rbp), %eax
    cltq
    movl    -12316(%rbp), %edx
    movslq  %edx, %rdx
    salq    $5, %rdx
    addq    %rdx, %rax
    movl    %ecx, -8208(%rbp,%rax,4)
    addl    $1, -12312(%rbp)
.L7:
    cmpl    $31, -12312(%rbp)
    jle .L8
    addl    $1, -12316(%rbp)
.L6:
    cmpl    $31, -12316(%rbp)
    jle .L9

    movl    $0, -12316(%rbp)
    jmp .L10
.L15:
    movl    $0, -12312(%rbp)
    jmp .L11
.L14:
    movl    $0, -12308(%rbp)
    jmp .L12
#APP
# 22 "matrMul.c" 1
    # LLVM-MCA-BEGIN foo
# 0 "" 2
#NO_APP
.L13:
    movl    -12312(%rbp), %eax
    cltq
    movl    -12316(%rbp), %edx
    movslq  %edx, %rdx
    salq    $5, %rdx
    addq    %rdx, %rax
    movl    -4112(%rbp,%rax,4), %edx
    movl    -12308(%rbp), %eax
    cltq
    movl    -12316(%rbp), %ecx
    movslq  %ecx, %rcx
    salq    $5, %rcx
    addq    %rcx, %rax
    movl    -12304(%rbp,%rax,4), %ecx
    movl    -12312(%rbp), %eax
    cltq
    movl    -12308(%rbp), %esi
    movslq  %esi, %rsi
    salq    $5, %rsi
    addq    %rsi, %rax
    movl    -8208(%rbp,%rax,4), %eax
    imull   %ecx, %eax
    addl    %eax, %edx
    movl    -12312(%rbp), %eax
    cltq
    movl    -12316(%rbp), %ecx
    movslq  %ecx, %rcx
```

```
    salq    $5, %rcx
    addq    %rcx, %rax
    movl    %edx, -4112(%rbp,%rax,4)
    addl    $1, -12308(%rbp)
.L12:
    cmpl    $31, -12308(%rbp)
    jle .L13
    addl    $1, -12312(%rbp)
.L11:
    cmpl    $31, -12312(%rbp)
    jle .L14
    addl    $1, -12316(%rbp)
.L10:
    cmpl    $31, -12316(%rbp)
    jle .L15
#APP
# 27 "matrMul.c" 1
    # LLVM-MCA-END
# 0 "" 2
#NO_APP
    movl    $0, %eax
    movq    -8(%rbp), %rdi
    xorq    %fs:40, %rdi
    je  .L17
    call    __stack_chk_fail@PLT
.L17:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE5:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0"
    .section    .note.GNU-stack,"",@progbits
```

Program run report:

```
0.56 msec task-clock                       #    0.518 CPUs utilized
                 0      context-switches    #    0.000 /sec
                 0      cpu-migrations      #    0.000 /sec
                54      page-faults         #   96.601 K/sec
         1,868,874      cycles              #    3.343 GHz
         2,286,744      instructions        #    1.22  insn per cycle
           259,939      branches            #  465.005 M/sec
             8,518      branch-misses       #    3.28% of all branches
           733,978      L1-dcache-loads     #    1.313 G/sec
<not counted>      L1-dcache-load-misses (0.00%)
<not counted>      LLC-loads             (0.00%)
<not counted>      LLC-load-misses       (0.00%)
       0.001079349 seconds time elapsed
       0.001167000 seconds user
       0.000000000 seconds sys
```

E Nvidia's GA100 vs. our $X3PU$

The evaluation for $X3PU$ is based on simulation and synthesis using Cadence environment:

- technology node: 7 nm
- area: 40 mm^2
- number of cells: 1024
- clock frequency: 1.275 GHz
- power: 5.12 W
- memory bus: 128 bis

while for GA100 GPU we used the spec [6]:

- technology node: 7 nm
- area: 846 mm^2
- number of cells: 6912
- clock frequency: 1.275 GHz
- power: 400 W
- memory bus: 5120 bits

For dense matrix multiplication on GA100 the information is provided by [5] (see Figure 7 and 8).

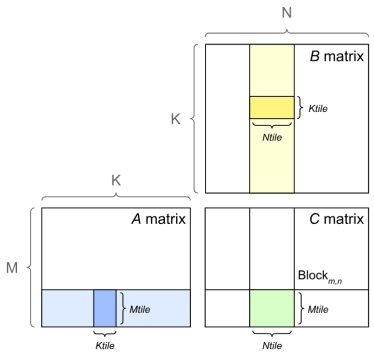

Figure 7: The definition for K, M, N [5].

According to Figure 7 and Figure 8 the matrix multiplication time for M=K=N on GA100 GPU is $t_{GPU}(1024) = 0.4ms$ According to simulation on a 1024 $X3PU$ the execution time for multiplying 1024×1024 matrices is $t_{X3PU}(1024) = 2.9ms$.

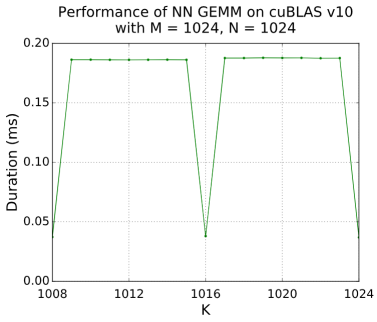

Figure 8: Execution time for matrix multiplication on GA100 [5].

Because the two ratio

$$\#cells_{GPU}/\#cells_{X3PU} = 6.75$$

and

$$t_{X3PU}(1024 \times 1024)/t_{GPU}(1024 \times 1024) = 7.25$$

are ~ 7, starting from

$$Power_{GPU}/Power_{X3PU} = 78$$

and

$$Area_{GPU}/Area_{X3PU} = 21$$

we conclude:

$X3PU$ performs $\sim 11\times$ computation for the same energy on a $\sim 3\times$ less area.