EE 789 : Assignment 2

Rohan Rajesh Kalbag
20D170033

November 2023

For all timing benchmarking, we compute the timing of performing matrix multiplication as

$$\Delta t = t_{mmul\ fin\ ack} - t_{mmul\ start\ ack}$$

the two timesteps can be extracted visually from the `.ghw` waveform. It was noticed that for all implementations discussed $t_{mmul\ start\ ack}$ was constant 46161 ns.

The python lambda `f = lambda x : (x - 46161)/1000` was used to get the timing in μs where x is $t_{mmul\ fin\ ack}$ in ns

1 Speeding Dot Product for Matrix Multiplication

We use the original implementation of dot product which performs a dot product of a 16 element. For this problem only the contents of `module[dot_product]` is changed and the rest of the code is left unchanged.

Implementation	**Time Taken (Δt) in μs**	**Speedup wrt Original**
Original	416.494	1
Unrolled by 4	170.734	2.439
Parallelized by 2 and Unrolled by 4	119.534	3.484

1.1 Original Implementation - `dot_product`

```
$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
    $branchblock[loop] {
        $dopipeline $depth 31 $fullrate
            $merge $entry $loopback
                $phi K := $zero<8> $on $entry nK $on $loopback
                $phi SUM := ($bitcast ($uint<32>) 0)  $on $entry nSUM $on $loopback
            $endmerge
            $volatile nK := (K + 1)
            $volatile continue_flag := (K < (ORDER-1))

            nSUM := (SUM + (A[I][K] * B[K][J]))
        $while continue_flag
    } (nSUM => result_exported)
    $volatile result := result_exported
}
```

1.2 Unrolled by 4 - `dot_product_unroll`

We open and perform **four** computations in each iteration, also reducing the number of iterations by four (we track them in *SUM0*, *SUM1*, *SUM2*, *SUM3*) and finally sum all of these together

```
$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
    $branchblock[loop] {

        $dopipeline $depth 31 $fullrate
            $merge $entry $loopback
                $phi K := $zero<8> $on $entry nK $on $loopback
                $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
                $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
                $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on $loopback
                $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on $loopback
            $endmerge
            $volatile nK := (K + 4)
            $volatile K1 := (K + 1)
            $volatile K2 := (K + 2)
            $volatile K3 := (K + 3)
            $volatile continue_flag := (K < (ORDER-4))

            nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
            nSUM1 := (SUM1 + (A[I][K1] * B[K1][J]))
            nSUM2 := (SUM2 + (A[I][K2] * B[K2][J]))
            nSUM3 := (SUM3 + (A[I][K3] * B[K3][J]))
        $while continue_flag
    } (nSUM0 => R0 nSUM1 => R1 nSUM2 => R2 nSUM3 => R3)
    $volatile result := ((R0 + R1) + (R2 + R3))
}
```

1.3 Parallelized by 2 and Unrolled by 4 - dot_product_parallel_unroll

In addition to performing unrolling, we split K into two disjoint parts one operating on first half and another operating on second half in parallel

```
$module [dot_product] $in (I J: $uint<8>) $out (result: $uint<32>) $is
{
    $parallelblock [computation]
    {
        $branchblock[loop0] {
            $dopipeline $depth 31 $fullrate
                $merge $entry $loopback
                    $phi K := $zero<8> $on $entry nK $on $loopback
                    $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
                    $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
                    $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on $loopback
                    $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on $loopback
                $endmerge
                $volatile nK := (K + 4)
                $volatile K1 := (K + 1)
                $volatile K2 := (K + 2)
                $volatile K3 := (K + 3)
                $volatile continue_flag := (K < (ORDER_BY_2-4))

                nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
                nSUM1 := (SUM1 + (A[I][K1] * B[K1][J]))
                nSUM2 := (SUM2 + (A[I][K2] * B[K2][J]))
                nSUM3 := (SUM3 + (A[I][K3] * B[K3][J]))
            $while continue_flag
        } (nSUM0 => R0 nSUM1 => R1 nSUM2 => R2 nSUM3 => R3)

        $branchblock[loop1] {
            $dopipeline $depth 31 $fullrate
                $merge $entry $loopback
                    $phi K := ($bitcast ($uint<8>) ORDER_BY_2) $on $entry nK $on $loopback
                    $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
                    $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
                    $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on $loopback
                    $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on $loopback
                $endmerge
                $volatile nK := (K + 4)
                $volatile K1 := (K + 1)
                $volatile K2 := (K + 2)
                $volatile K3 := (K + 3)
                $volatile continue_flag := (K < (ORDER-4))

                nSUM0 := (SUM0 + (A[I][K] * B[K][J]))
                nSUM1 := (SUM1 + (A[I][K1] * B[K1][J]))
                nSUM2 := (SUM2 + (A[I][K2] * B[K2][J]))
                nSUM3 := (SUM3 + (A[I][K3] * B[K3][J]))
            $while continue_flag
        } (nSUM0 => R4 nSUM1 => R5 nSUM2 => R6 nSUM3 => R7)
    }(R0 => X0 R1 => X1 R2 => X2 R3 => X3 R4 => X4 R5 => X5 R6 => X6 R7 => X7)

    $volatile result := (((X0 + X1) + (X2 + X3)) + ((X4 + X5) + (X6 + X7)))

}
```

2 Parallelize Matrix Multiplication into 4 8X8 blocks

We divide the matrix A and B into four parts of sizes 8×8 and then distribute computations of smaller matrix multiplications in parallel. We add x and y offsets to the dot product module and the accordingly the computed sum is stored into C.

$$A_{16\times16} = \begin{bmatrix} M_{8\times8} & N_{8\times8} \\ O_{8\times8} & D_{8\times8} \end{bmatrix}$$

$$B_{16\times16} = \begin{bmatrix} E_{8\times8} & F_{8\times8} \\ G_{8\times8} & H_{8\times8} \end{bmatrix}$$

$$C_{16\times16} = \begin{bmatrix} M\cdot E + N\cdot G & M\cdot F + N\cdot H \\ O\cdot E + D\cdot G & O\cdot F + D\cdot H \end{bmatrix}$$

Implementation	**Time Taken (Δt) in μs**	**Speedup wrt Vanilla**
Vanilla	419.964	1
Unrolled by 4	174.204	2.41
Parallelized by 2 and Unrolled by 4	133.244	3.151

2.1 Common mmul module

Performs computation of each 8×8 submatrix of C in parallelblock of individual branchblocks

```
$module [mmul] $in () $out () $is
{

        $parallelblock [computation]
        {
                $branchblock[meng] {
                        $merge $entry I_loopback
                                $phi I := $zero<8> $on $entry nI $on I_loopback
                        $endmerge

                        $volatile nI := (I + 1)

                        $dopipeline $depth 31 $fullrate
                                $merge $entry $loopback
                                        $phi J := $zero<8> $on $entry nJ $on $loopback
                                $endmerge
                                $volatile nJ := (J + 1)
                                $volatile continue_flag := (J < (ORDER_BY_2 - 1))

                                ME := ($call dot_product (I J $zero<8> $zero<8>))
                                NG := ($call dot_product (I J offset offset))
                                C[I][J] := (ME + NG)

                        $while continue_flag

                        $if (I < (ORDER_BY_2 - 1)) $then $place [I_loopback] $endif
                }
```

```
$branchblock[mfnh] {
        $merge $entry I_loopback
                $phi I := $zero<8> $on $entry nI $on I_loopback
        $endmerge

        $volatile nI := (I + 1)

        $dopipeline $depth 31 $fullrate
                $merge $entry $loopback
                        $phi J := $zero<8> $on $entry nJ $on $loopback
                $endmerge
                $volatile nJ := (J + 1)
                $volatile continue_flag := (J < (ORDER_BY_2 - 1))
                $volatile offsetJ := (J + offset)

                MF := ($call dot_product (I offsetJ $zero<8> $zero<8>))
                NH := ($call dot_product (I offsetJ offset offset))
                C[I][offsetJ] := (MF + NH)

        $while continue_flag

        $if (I < (ORDER_BY_2 - 1)) $then $place [I_loopback] $endif
}

$branchblock[oedg] {
        $merge $entry I_loopback
                $phi I := $zero<8> $on $entry nI $on I_loopback
        $endmerge

        $volatile nI := (I + 1)

        $dopipeline $depth 31 $fullrate
                $merge $entry $loopback
                        $phi J := $zero<8> $on $entry nJ $on $loopback
                $endmerge
                $volatile nJ := (J + 1)
                $volatile continue_flag := (J < (ORDER_BY_2 - 1))
                $volatile offsetI := (I + offset)

                OE := ($call dot_product (offsetI J $zero<8> $zero<8>))
                DG := ($call dot_product (offsetI J offset offset))
                C[offsetI][J] := (OE + DG)

        $while continue_flag

        $if (I < (ORDER_BY_2 - 1)) $then $place [I_loopback] $endif
}
```

```
                $branchblock[ofdh] {
                        $merge $entry I_loopback
                                $phi I := $zero<8> $on $entry nI $on I_loopback
                        $endmerge

                        $volatile nI := (I + 1)

                        $dopipeline $depth 31 $fullrate
                                $merge $entry $loopback
                                        $phi J := $zero<8> $on $entry nJ $on $loopback
                                $endmerge
                                $volatile nJ := (J + 1)
                                $volatile continue_flag := (J < (ORDER_BY_2 - 1))
                                $volatile offsetI := (I + offset)
                                $volatile offsetJ := (J + offset)

                                OF := ($call dot_product (offsetI offsetJ $zero<8> $zero<8>))
                                DH := ($call dot_product (offsetI offsetJ offset offset))
                                C[offsetI][offsetJ] := (OF + DH)

                        $while continue_flag

                        $if (I < (ORDER_BY_2 - 1)) $then $place [I_loopback] $endif
                }
        }
}
```

2.2 Vanilla 8×8 dot product - `divide_into_four_impl`

Here OF1 and OF2 are the offsets for the x and y values used for positioning while `$call`

```
$module [dot_product] $in (I J OF1 OF2: $uint<8>) $out (result: $uint<32>) $is
{
        $branchblock[loop] {

                $dopipeline $depth 31 $fullrate
                        $merge $entry $loopback
                                $phi K := $zero<8> $on $entry nK $on $loopback
                                $phi SUM := ($bitcast ($uint<32>) 0)  $on $entry nSUM $on $loopback
                        $endmerge
                        $volatile nK := (K + 1)
                        $volatile continue_flag := (K < (ORDER_BY_2-1))
                        $volatile KX := (K + OF1)
                        $volatile KY := (K + OF2)

                        nSUM := (SUM + (A[I][KX] * B[KY][J]))
                $while continue_flag
        } (nSUM => result_exported)
        $volatile result := result_exported
}
```

2.3 Unrolled by 4 8×8 dot product - `divide_into_four_impl_unroll`

We open and perform **four** computations in each iteration, also reducing the number of iterations by four (we track them in *SUM0*, *SUM1*, *SUM2*, *SUM3*) and finally sum all of these together

```
$module [dot_product] $in (I J OF1 OF2: $uint<8>) $out (result: $uint<32>) $is
{
        $branchblock[loop] {

                $dopipeline $depth 31 $fullrate
                        $merge $entry $loopback
                                $phi K := $zero<8> $on $entry nK $on $loopback
                                $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on $loopback
                                $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on $loopback
                                $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on $loopback
                                $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on $loopback
                        $endmerge

                        $volatile continue_flag := (K < (ORDER_BY_2-4))

                        $volatile nK := (K + 4)
                        $volatile K1 := (K + 1)
                        $volatile K2 := (K + 2)
                        $volatile K3 := (K + 3)

                        $volatile K0X := (K + OF1)
                        $volatile K1X := (K1 + OF1)
                        $volatile K2X := (K2 + OF1)
                        $volatile K3X := (K3 + OF1)
                        $volatile K0Y := (K + OF2)
                        $volatile K1Y := (K1 + OF2)
                        $volatile K2Y := (K2 + OF2)
                        $volatile K3Y := (K3 + OF2)

                        nSUM0 := (SUM0 + (A[I][K0X] * B[K0Y][J]))
                        nSUM1 := (SUM1 + (A[I][K1X] * B[K1Y][J]))
                        nSUM2 := (SUM2 + (A[I][K2X] * B[K2Y][J]))
                        nSUM3 := (SUM3 + (A[I][K3X] * B[K3Y][J]))
                $while continue_flag
        } (nSUM0 => R0 nSUM1 => R1 nSUM2 => R2 nSUM3 => R3)
        $volatile result := ((R0 + R1) + (R2 + R3))
}
```

2.4 Parallelized by 2 and Unrolled by 4 8×8 dot product - divide_into_four_impl_parallel_unroll

In addition to performing unrolling, we split K into two disjoint parts one operating on first half and another operating on second half in parallel.

```
$module [dot_product] $in (I J OF1 OF2: $uint<8>) $out (result: $uint<32>) $is
{
        $parallelblock [computation]
        {
                $branchblock[loop0] {
                        $dopipeline $depth 31 $fullrate
                                $merge $entry $loopback
                                        $phi K := $zero<8> $on $entry nK $on $loopback
                                        $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on
                                        ↪  $loopback
                                        $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on
                                        ↪  $loopback
                                        $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on
                                        ↪  $loopback
                                        $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on
                                        ↪  $loopback
                                $endmerge

                                $volatile continue_flag := (K < (ORDER_BY_4-4))

                                $volatile nK := (K + 4)
                                $volatile K1 := (K + 1)
                                $volatile K2 := (K + 2)
                                $volatile K3 := (K + 3)

                                $volatile K0X := (K + OF1)
                                $volatile K1X := (K1 + OF1)
                                $volatile K2X := (K2 + OF1)
                                $volatile K3X := (K3 + OF1)
                                $volatile K0Y := (K + OF2)
                                $volatile K1Y := (K1 + OF2)
                                $volatile K2Y := (K2 + OF2)
                                $volatile K3Y := (K3 + OF2)

                                nSUM0 := (SUM0 + (A[I][K0X] * B[K0Y][J]))
                                nSUM1 := (SUM1 + (A[I][K1X] * B[K1Y][J]))
                                nSUM2 := (SUM2 + (A[I][K2X] * B[K2Y][J]))
                                nSUM3 := (SUM3 + (A[I][K3X] * B[K3Y][J]))
                        $while continue_flag
                } (nSUM0 => R0 nSUM1 => R1 nSUM2 => R2 nSUM3 => R3)
```

```
        $branchblock[loop1] {
                $dopipeline $depth 31 $fullrate
                        $merge $entry $loopback
                                $phi K := ($bitcast ($uint<8>) ORDER_BY_4) $on $entry nK $on
                                ↪  $loopback
                                $phi SUM0 := ($bitcast ($uint<32>) 0)  $on $entry nSUM0 $on
                                ↪  $loopback
                                $phi SUM1 := ($bitcast ($uint<32>) 0)  $on $entry nSUM1 $on
                                ↪  $loopback
                                $phi SUM2 := ($bitcast ($uint<32>) 0)  $on $entry nSUM2 $on
                                ↪  $loopback
                                $phi SUM3 := ($bitcast ($uint<32>) 0)  $on $entry nSUM3 $on
                                ↪  $loopback
                        $endmerge

                        $volatile continue_flag := (K < (ORDER_BY_2-4))

                        $volatile nK := (K + 4)
                        $volatile K1 := (K + 1)
                        $volatile K2 := (K + 2)
                        $volatile K3 := (K + 3)

                        $volatile K0X := (K + OF1)
                        $volatile K1X := (K1 + OF1)
                        $volatile K2X := (K2 + OF1)
                        $volatile K3X := (K3 + OF1)
                        $volatile K0Y := (K + OF2)
                        $volatile K1Y := (K1 + OF2)
                        $volatile K2Y := (K2 + OF2)
                        $volatile K3Y := (K3 + OF2)

                        nSUM0 := (SUM0 + (A[I][K0X] * B[K0Y][J]))
                        nSUM1 := (SUM1 + (A[I][K1X] * B[K1Y][J]))
                        nSUM2 := (SUM2 + (A[I][K2X] * B[K2Y][J]))
                        nSUM3 := (SUM3 + (A[I][K3X] * B[K3Y][J]))
                $while continue_flag
        } (nSUM0 => R4 nSUM1 => R5 nSUM2 => R6 nSUM3 => R7)
    }(R0 => X0 R1 => X1 R2 => X2 R3 => X3 R4 => X4 R5 => X5 R6 => X6 R7 => X7)

    $volatile result := (((X0 + X1) + (X2 + X3)) + ((X4 + X5) + (X6 + X7)))
}
```

3 Matrix Multiplication as sum of rank-1 Matrices

We perform computation of the matrix product by keeping a running sum of the outer products of columns of A and rows of B. This is done using a main outer loop for the row/column number (K). Then inner loops for I, J where $C[I][J]$ is updated using $A[I][K] * B[K][J]$

$$A = [c_1\ c_2\ c_3\ \ \ldots],\ B = [r_1\ r_2\ r_3\ \ \ldots],\ c_i \in R^{n\times 1},\ r_i \in R^{1\times n}$$

$$C = \sum_{i=1}^{16} c_i * r_i,\ c_i * r_i \in R^{n\times n}$$

Implementation	Time Taken (Δt) in μs	Speedup wrt Vanilla
Vanilla	317.944	1
Unrolled by 4	169.454	1.876
Parallelized by 2 and Unrolled by 4	133.424	2.382

3.1 Vanilla Implementation - sum_of_outer_products

```
$module [mmul] $in () $out () $is
{
        $branchblock[loop] {
                $merge $entry K_loopback
                        $phi K := $zero<8> $on $entry nK $on K_loopback
                $endmerge

                        $merge $entry I_loopback
                                $phi I := $zero<8> $on $entry nI $on I_loopback
                        $endmerge

                                $dopipeline $depth 31 $buffering 4
                                $fullrate

                                $merge $entry $loopback
                                        $phi J := $zero<8> $on $entry nJ $on $loopback
                                $endmerge
                                $volatile nJ := (J + 1)
                                $volatile Jcontinue_flag := (J < (ORDER - 1))
                                $volatile prev_C := C[I][J]

                                C[I][J] := (prev_C + (A[I][K] * B[K][J]))

                                $while Jcontinue_flag

                        $volatile nI := (I + 1)
                        $volatile Icontinue_flag := (I < (ORDER - 1))

                        $if Icontinue_flag $then $place [I_loopback] $endif

                $volatile nK := (K + 1)
                $volatile Kcontinue_flag := (K < (ORDER - 1))

                $if Kcontinue_flag $then $place [K_loopback] $endif
        }
}
```

3.2 Unrolled by 4 - sum_of_outer_products_unroll

We open and perform **four** computations in each J iteration reducing iterations by three-fourths.

```
$module [mmul] $in () $out () $is
{
        $branchblock[loop] {
                $merge $entry K_loopback
                        $phi K := $zero<8> $on $entry nK $on K_loopback
                $endmerge

                        $merge $entry I_loopback
                                $phi I := $zero<8> $on $entry nI $on I_loopback
                        $endmerge

                                $dopipeline $depth 31 $buffering 4
                                $fullrate

                                $merge $entry $loopback
                                        $phi J := $zero<8> $on $entry nJ $on $loopback
                                $endmerge

                                $volatile nJ := (J + 4)

                                $volatile J1 := (J + 1)
                                $volatile J2 := (J + 2)
                                $volatile J3 := (J + 3)

                                $volatile Jcontinue_flag := (J < (ORDER - 4))

                                prev_C := C[I][J]
                                prev_C1 := C[I][J1]
                                prev_C2 := C[I][J2]
                                prev_C3 := C[I][J3]
                                lhs := A[I][K]

                                C[I][J] := (prev_C + (lhs * B[K][J]))
                                C[I][J1] := (prev_C1 + (lhs * B[K][J1]))
                                C[I][J2] := (prev_C2 + (lhs * B[K][J2]))
                                C[I][J3] := (prev_C3 + (lhs * B[K][J3]))

                                $while Jcontinue_flag

                        $volatile nI := (I + 1)
                        $volatile Icontinue_flag := (I < (ORDER - 1))

                        $if Icontinue_flag $then $place [I_loopback] $endif

                $volatile nK := (K + 1)
                $volatile Kcontinue_flag := (K < (ORDER - 1))

                $if Kcontinue_flag $then $place [K_loopback] $endif
        }
}
```

3.3 Parallelized by 2 and Unrolled by 4 - sum_of_outer_products_parallel_unroll

In addition to performing unrolling, we split K into **two** disjoint parts one operating on first half and another operating on second half in parallel to further improve speedup.

```
$module [mmul] $in () $out () $is
{
        $parallelblock [computation]
        {

                $branchblock[loop0] {
                        $merge $entry K_loopback
                                $phi K := $zero<8> $on $entry nK $on K_loopback
                        $endmerge

                                $merge $entry I_loopback
                                        $phi I := $zero<8> $on $entry nI $on I_loopback
                                $endmerge

                                        $dopipeline $depth 31 $buffering 4
                                        $fullrate

                                        $merge $entry $loopback
                                                $phi J := $zero<8> $on $entry nJ $on $loopback
                                        $endmerge

                                        $volatile nJ := (J + 4)

                                        $volatile J1 := (J + 1)
                                        $volatile J2 := (J + 2)
                                        $volatile J3 := (J + 3)

                                        $volatile Jcontinue_flag := (J < (ORDER - 4))

                                        prev_C := C[I][J]
                                        prev_C1 := C[I][J1]
                                        prev_C2 := C[I][J2]
                                        prev_C3 := C[I][J3]
                                        lhs := A[I][K]

                                        C[I][J] := (prev_C + (lhs * B[K][J]))
                                        C[I][J1] := (prev_C1 + (lhs * B[K][J1]))
                                        C[I][J2] := (prev_C2 + (lhs * B[K][J2]))
                                        C[I][J3] := (prev_C3 + (lhs * B[K][J3]))

                                        $while Jcontinue_flag

                                $volatile nI := (I + 1)
                                $volatile Icontinue_flag := (I < (ORDER - 1))

                                $if Icontinue_flag $then $place [I_loopback] $endif

                        $volatile nK := (K + 2)
                        $volatile Kcontinue_flag := (K < (ORDER - 2))

                        $if Kcontinue_flag $then $place [K_loopback] $endif
                }
```

```
			$branchblock[loop1] {
				$merge $entry K_loopback
					$phi K := ($bitcast ($uint<8>) 1) $on $entry nK $on K_loopback
				$endmerge

					$merge $entry I_loopback
						$phi I := $zero<8> $on $entry nI $on I_loopback
					$endmerge

						$dopipeline $depth 31 $buffering 4
						$fullrate

						$merge $entry $loopback
							$phi J := $zero<8> $on $entry nJ $on $loopback
						$endmerge

						$volatile nJ := (J + 4)

						$volatile J1 := (J + 1)
						$volatile J2 := (J + 2)
						$volatile J3 := (J + 3)

						$volatile Jcontinue_flag := (J < (ORDER - 4))

						prev_C := C[I][J]
						prev_C1 := C[I][J1]
						prev_C2 := C[I][J2]
						prev_C3 := C[I][J3]
						lhs := A[I][K]

						C[I][J] := (prev_C + (lhs * B[K][J]))
						C[I][J1] := (prev_C1 + (lhs * B[K][J1]))
						C[I][J2] := (prev_C2 + (lhs * B[K][J2]))
						C[I][J3] := (prev_C3 + (lhs * B[K][J3]))

						$while Jcontinue_flag

					$volatile nI := (I + 1)
					$volatile Icontinue_flag := (I < (ORDER - 1))

					$if Icontinue_flag $then $place [I_loopback] $endif

				$volatile nK := (K + 2)
				$volatile Kcontinue_flag := (K < (ORDER - 1))

				$if Kcontinue_flag $then $place [K_loopback] $endif
			}
		}
}
```