Exploiting Multi-Level Parallelism in Streaming Applications for Heterogeneous Platforms with GPUs

Balevic, A.

Appendix A

Application Source Codes

A.1 Predictor

```
1  int produce();
2  int predict(int um, int ml);
3  void consume(int a);
4
5  #pragma compaan_procedure predictor
6  void predictor(void)
7  {
8     int a[5][5];
9     int i,j;
10
11    for (i = 0; i < 4; i = i + 1)
12       for (j = 0; j < 4; j = j + 1)
13          a[i][j] = produce();                      //Statement P
14
15    for (i = 1; i <= 4; i = i + 1)
16       for (j = 1; j <= 4; j = j + 1)
17          a[i][j] = predict(a[i-1][j], a[i][j-1]);  //Statement T
18
19    for (i = 1; i <= 4; i = i + 1)
20       for (j = 1; j <= 4; j = j + 1)
21          consume(a[i][j]);                         //Statement C
22 }
```

Listing A.1: Predictor SANLP

A.2 Grid

```
1  int produce();
2  int grid(int a);
3  void consume(int a);
4
5  #pragma compaan_procedure grid
6  void grid(void)
7  {
8     int a[10];
9     int i,j,x,y;
10
11    for (x = 0; x < 10; x = x + 1)
12          a[x] = produce();              //Statement P
13
14    for (i = 1; i <= 4; i = i + 1)
15       for (j = 1; j <= 4; j = j + 1)
16          a[i+j] = grid(a[i+j]);         //Statement T
17
18    for (y = 2; y < 8; y = y + 1)
19          consume(a[y]);                 //Statement C
20 }
```

Listing A.2: Grid SANLP

A.3 Sobel

```
1
2  #define MAX_WIDTH 250
3  #define MAX_HEIGHT 250
4
5  #define abs(x) ( (x < 0) ? -(x) : (x) )
6
7  void readPixel(int * p);
8  void writePixel(int value);
9
10 void gradient(const int a1, const int a2, const int a3, const int a4,
11   const int a5, const int a6, int* out);
12
13 void absVal(const int x, const int y, int* out);
14
15 #pragma compaan_procedure sobel
16 void sobel(
17 #pragma compaan_parameter 10 25
18   int M,
19 #pragma compaan_parameter 10 25
20   int N, int image_in[MAX_WIDTH][MAX_HEIGHT],
21   int image_out[MAX_WIDTH][MAX_HEIGHT])
```

```
22 {
23
24  int i, j;
25  int image[M][N];
26  int Jx[M][N];
27  int Jy[M][N];
28  int av[M][N];
29
30  for (j = 0; j < M; j++)
31   for (i = 0; i < N; i++)
32    image[j][i] = image_in[j][i];
33
34  for (j = 1; j < M - 1; j++) {
35   for (i = 1; i < N - 1; i++) {
36    gradient(image[j - 1][i - 1], image[j][i - 1], image[j + 1][i - 1],
37      image[j - 1][i + 1], image[j][i + 1], image[j + 1][i + 1],
38      &(Jx[j][i]));
39    gradient(image[j - 1][i - 1], image[j - 1][i], image[j - 1][i + 1],
40      image[j + 1][i - 1], image[j + 1][i], image[j + 1][i + 1],
41      &(Jy[j][i]));
42    absVal(Jx[j][i], Jy[j][i], &(av[j][i]));
43   }
44  }
45
46  for (j = 1; j < M - 1; j++)
47   for (i = 1; i < N - 1; i++)
48    image_out[j][i] = av[j][i];
49
50 }
```

Listing A.3: Sobel SANLP

Appendix B

KPN2GPU

B.1 Predictor

B.1.1 PPN Model

Default PPN (Compaan)

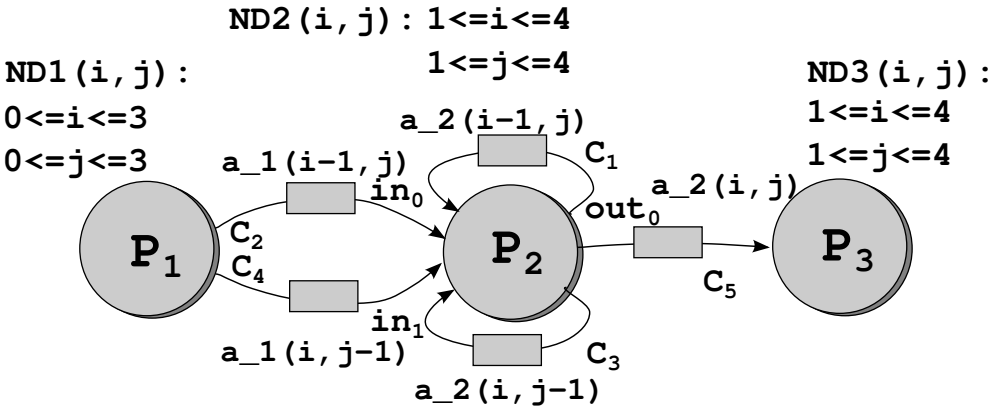
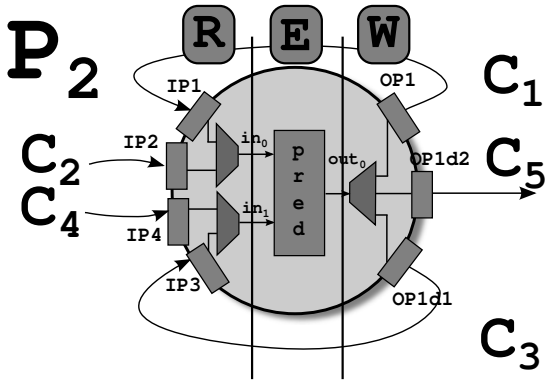


Figure B.1: The PPN obtained from the source code in Listing A.1.

Specification of Process P_2

B.1.2 Node P_2: Space-Time Mapping

- schedule: $t(P_2, (i, j)) = i + j - 2$



$$D_{P_2} = \{(i,j) \mid 1 \le i \le 4 \wedge 1 \le j \le 4\}$$
$$\text{IPD1} = D_T \cap \{(i,j) \mid i \ge 2\}$$
$$\text{IPD2} = D_T \cap \{(i,j) \mid i = 1\}$$
$$\text{IPD3} = D_T \cap \{(i,j) \mid j \ge 2\}$$
$$\text{IPD4} = D_T \cap \{(i,j) \mid j = 1\}$$
$$\text{OPD1} = D_T \cap \{(i,j) \mid i \le 3\}$$
$$\text{OPD1d1} = D_T \cap \{(i,j) \mid j \le 3\}$$
$$\text{OPD1d2} = D_T$$
$$M_{C1} = M_{C2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$
$$M_{C3} = M_{C4} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$$

Figure B.2: (a) Process P_2, (b) Specification of P_2's domain, port domains, and mapping matrices of incoming channels C_1-C_4.

- allocation: $p(P_2, (i,j)) = j$

Resulting space-time mapping (schedule + allocation): $T_{P_2} = \begin{bmatrix} 1 & 1 & -2 \\ 0 & 1 & 0 \end{bmatrix}$

Inverse space-time mapping: $T_{P_2}^{-1} = \begin{bmatrix} 1 & -1 & 2 \\ 0 & 1 & 0 \end{bmatrix}$

Coordinate system transformation: $\begin{bmatrix} i \\ j \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & -1 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t0 \\ p0 \\ 1 \end{bmatrix}$

Iterators expressed in space-time coordinates: $i = t0 - p0 + 2$, $j = p0$.

- Width $W = p0_{max} = 4$: Default num threads for the allocation
- Depth $D = t0_{max} = 7$: Default num time steps for the schedule

B.1.3 Node P_2: DPV Components

Parallel Node Domain

$D_{P_2}{}^{\|} = \{(p0, t0) \mid (t0 - p0 + 1 \ge 0) \wedge (-t0 + p0 + 2 \ge 0) \wedge (p0 - 1 >= 0) \wedge (-p0 + 4 >= 0)\}$

Port Domains

$$\begin{aligned}
\text{IPD1} &= D_T{}^{\|} \cap \{(p0, t0) \mid t0 - p0 \geq 0\} \\
\text{IPD2} &= D_T{}^{\|} \cap \{(p0, t0) \mid t0 - p0 - 1 = 0\} \\
\text{IPD3} &= D_T{}^{\|} \cap \{(p0, t0) \mid p0 - 2 \geq 0\} \\
\text{IPD4} &= D_T{}^{\|} \cap \{(p0, t0) \mid p0 - 1 = 0\} \\
\text{OPD1} &= D_T{}^{\|} \cap \{(p0, t0) \mid -t0 + p0 + 1 \geq 0\} \\
\text{OPD1d1} &= D_T{}^{\|} \cap \{(p0, t0) \mid -p0 + 3 \geq 0\} \\
\text{OPD1d2} &= D_T{}^{\|}
\end{aligned}$$

Data Parallel Channels

Input channels for argument in_0

DPC1 Input channel for argument in_0, when read from P'_2. Derived from PPN self-link C_1 with mapping M_{C_1}.

- Recall : $M_{C_1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$
- DPC1 Mapping: $M'_{C_1} = T_{P_2} M_{C_1} T_{P_2}^{-1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC1 Array: $\mathtt{a_2}[7][4]$
- Read access: $\mathtt{in0} \leftarrow \mathtt{a_2}(\mathtt{t0} - 1, \mathtt{p0})$

DPC2 Input channel for argument in_0, when read from P'_1. Derived from external PPN channel C_2 with mapping M_{C_2}.

- Recall : $M_{C_2} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$
- DPC2 Mapping: $M'_{C_2} = T_{P_1} M_{C_2} T_{P_2}^{-1}$, where $T_{P_1} = I$.
- DPC2 Mapping: $M'_{C_2} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC2 Array: $\mathtt{a_1}[5][5]$
- Read access: $\mathtt{in0} \leftarrow \mathtt{a_1}(\mathtt{t0} - \mathtt{p0} + 1, \mathtt{p0})$

Input channels for argument in_1

DPC3 Input channel for argument in_1, when read from P'_2. Derived from PPN self-link C_3 with mapping M_{C_3}. Implemented as array $\mathtt{a_2}$.

- Recall : $M_{C_3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$
- DPC3 Mapping: $M'_{C_3} = T_{P_2} M_{C_3} T_{P_2}^{-1} = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC3 Array: $\mathtt{a_2[5][5]}$
- Read access: $\mathtt{in1} \leftarrow \mathtt{a_2(t0 - 1, p0 - 1)}$

DPC4 Input channel for argument in_1, when read from P'_1. Derived from external PPN channel C_4 with mapping M_{C_4}.

- Recall : $M_{C_4} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{bmatrix}$
- DPC2 Mapping: $M'_{C_4} = T_{P_1} M_{C_4} T_{P_2}^{-1}$, where $T_{P_1} = I$.
- DPC2 Mapping: $M'_{C_4} = \begin{bmatrix} 1 & -1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$
- DPC4 Array: $\mathtt{a_1[5][5]}$
- Read access: $\mathtt{in0} \leftarrow \mathtt{a_1(t0 - p0 + 1, p0)}$

Output channels for argument out_0

DPC5 External output channel for argument out_0 produced by DPP P'_2, derived from PPN channel C_5.

- DPC5 Array: $\mathtt{a_2}$
- Write access: $\mathtt{out0} \leftarrow \mathtt{a_2(t0, p0)}$

DPC1, DPC3 Self-links feeding argument out_0 to the next iteration of DPP P'_2. See the specification above.

B.1.4 Predictor: Host Code

```
1  void runTest( int argc, char** argv){
2     //Initialization
3     printf("Allocating␣memory␣on␣CPU...\n");
4     int* a_1 = (int*) malloc(ga_1_MEM_SIZE);
5     int* a_2 = (int*) malloc(ga_2_MEM_SIZE);
6
7     printf("Allocating␣memory␣on␣GPU...\n");
8     int* ga_1; cudaMalloc((void**) &ga_1, ga_1_MEM_SIZE);
9     int* ga_2; cudaMalloc((void**) &ga_2, ga_2_MEM_SIZE);
10
11    // Kernel Configuration
12    int ND_1_GRIDDim = 1; int ND_1_TBDim = ND_1_W;
13    int ND_2_GRIDDim = 1; int ND_2_TBDim = ND_2_W;
14    int ND_3_GRIDDim = 1; int ND_r_TBDim = ND_3_W;
15
16    // Kernel Calls
17    ND_1_Kernel<<< ND_1_GRIDDim, ND_1_TBDim >>>(ga_1);
18
19    ND_2_Kernel<<< ND_2_GRIDDim, ND_2_TBDim >>>(ga_1, ga_2);
20
21    ND_3_Kernel<<< ND_3_GRIDDim, ND_3_TBDim >>>(ga_2);
22
23    cudaMemcpy(a_1, ga_1, ga_1_MEM_SIZE, cudaMemcpyDeviceToHost);
24    cudaMemcpy(a_2, ga_2, ga_2_MEM_SIZE, cudaMemcpyDeviceToHost);
25
26   //Clean up
27    free(a_1);
28    free(a_2);
29
30    cudaFree(ga_1);
31    cudaFree(ga_2);
32
33    cudaThreadExit();
34 }
```

Listing B.1: Generated CUDA host code.

B.1.5 Node P_2': CUDA Kernel (Default)

```
1 __host__ __device__ int predict(int um, int ml);
2 #define ND_2_W (4)
3 #define ND_2_D (7)
4
5 #define ACTIVE_ND_2 ((t0-p0+1 >= 0) && (-t0+p0+2 >= 0) && (p0-1 >= 0) && (-p0
      +4 >= 0))
```

```
 6
 7 #define ND_2IP_1 ((ACTIVE_ND_2) && (((t0-p0 >= 0))))
 8 #define ND_2IP_2 ((ACTIVE_ND_2) && (((t0-p0+1 == 0))))
 9 #define ND_2IP_3 ((ACTIVE_ND_2) && (((p0-2 >= 0))))
10 #define ND_2IP_4 ((ACTIVE_ND_2) && (((p0-1 == 0))))
11
12 #define ND_2OP_1 ((ACTIVE_ND_2) && (((-t0+p0+1 >= 0))))
13 #define ND_2OP_1_d1 ((ACTIVE_ND_2) && (((-p0+3 >= 0))))
14 #define ND_2OP_1_d2 (ACTIVE_ND_2)
15
16 #define a_2_stride ND_2_W
17 #define a_1_stride ND_1_W
18 #define DPC1(t,p) ga_2[ a_2_stride * (t-1) + (p) ]
19 #define DPC2(t,p) ga_1[ a_1_stride * (t-p+1) + (p) ]
20 #define DPC3(t,p) ga_2[ a_2_stride * (t-1) + (p-1) ]
21 #define DPC4(t,p) ga_1[ a_1_stride * (t-p+2) + (p-1) ]
22 #define DPC5(t,p) ga_2[ a_2_stride * (t) + (p) ]
23
24 __global__ void ND_2_Kernel( int *ga_1, //input channel
25                              int *ga_2 //input-output channel
26                            )
27 {
28    int in_0;
29    int in_1;
30    int out_0;
31
32    // Mapping: CUDA Threads to Processing Entities
33  // threadIdx.x - unique thread identifier
34    int p0 = ((threadIdx.x) + ((1)));
35
36    // Number of synchronous time steps in PND
37    for(int t0 = 0; t0 < ND_2_D; t0++)
38    {
39       ////////////////////////////////////
40       // Process Iteration t0
41       ////////////////////////////////////
42
43       ////////////////////////////////////
44       // Phase I: READ
45       ////////////////////////////////////
46       if (ND_2IP_1) {
47              in_0 = DPC1(t0, p0);
48       }
49       if (ND_2IP_2) {
50              in_0 = DPC2(t0, p0);
51       }
52       if (ND_2IP_3) {
53              in_1 = DPC3(t0, p0);
54       }
```

```
55        if (ND_2IP_4) {
56              in_1 = DPC4(t0, p0);
57        }
58
59        ////////////////////////////////////
60        // Phase II: EXECUTE
61        ////////////////////////////////////
62        if (ACTIVE_ND_2) {
63              out_0 = predictor(in_0, in_1);
64        }
65
66        ////////////////////////////////////
67        // Phase III: WRITE
68        // (for each output arg - 1 write per memory array!)
69        ////////////////////////////////////
70        if ((ND_2OP_1) || (ND_2OP_1_d1) || (ND_2OP_1_d2)) {
71              DPC5(t0, p0) = out_0;
72        }
73        __syncthreads();
74
75    } //end for
76
77 } //end ND_2_Kernel
```

Listing B.2: CUDA kernel: default.

B.1.6 Node P_2': CUDA Kernel (Optimized)

```
1  //Channel width
2  #define sa_2_stride (4)
3
4  //(a) Default buffer size
5  #define DPC13_SIZE (7*4)
6  #define DPC1(t,p) sa_2[ sa_2_stride * (t-1) + (p) ]
7  #define DPC3(t,p) sa_2[ sa_2_stride * (t-1) + (p-1) ]
8  #define DPC13(t,p) sa_2[ sa_2_stride * (t) + (p) ]
9
10 //(b) Optimized buffer size
11 #define DPC13_SIZE (1*4)
12 #define DPC1(t,p) sa_2[ p ]
13 #define DPC3(t,p) sa_2[ p-1 ]
14 #define DPC13(t,p) sa_2[ p  ]
15
16 __global__ void ND_2_Kernel( int *ga_1, //input channel
17                              int *ga_2 //input-output channel
18                            )
19 {
20    __shared__ int sa_2[DPC13_SIZE];      //self-links
```

```
21
22    int in_0;
23    int in_1;
24    int out_0;
25
26    // Mapping of virtual processors to CUDA threads
27    int p0 = ((threadIdx.x) + ((1)));
28
29    for(int t0 = 0; t0 < ND_2_D; t0++)
30    {
31       ////////////////////////////////////
32       // Process Iteration t0
33       ////////////////////////////////////
34
35       ////////////////////////////////////
36       // Phase I: READ
37       ////////////////////////////////////
38       if (ND_2IP_1) {
39              in_0 = DPC1(t0, p0);
40       }
41       if (ND_2IP_2) {
42              in_0 = DPC2(t0, p0);
43       }
44       if (ND_2IP_3) {
45              in_1 = DPC3(t0, p0);
46       }
47       if (ND_2IP_4) {
48              in_1 = DPC4(t0, p0);
49       }
50       __syncthreads();
51
52       ////////////////////////////////////
53       // Phase II: EXECUTE
54       ////////////////////////////////////
55       if (ACTIVE_ND_2) {
56              out_0 = predictor(in_0, in_1);
57       }
58
59       ////////////////////////////////////
60       // Phase III: WRITE
61       // (for each output arg - 1 write per memory array!)
62       ////////////////////////////////////
63       // Only writes to self-links must complete before
64       // the first threads start read phase of the next process iteration
65       if ((ND_2OP_1) || (ND_2OP_1_d1)) {
66              DPC13(t0, p0) = out_0;
67       }
68       __syncthreads();
69
```

```
70        if (ND_2OP_1_d2) {
71              DPC5(t0, p0) = out_0;
72        }
73
74    } //end for
75
76 } //end ND_2_Kernel
```

Listing B.3: CUDA kernel: optimized.

B.1.7 Scaling

Adjusted Host Code

Given the data parallel tile domain in space-time coordinates $(p1, t1)$ with maximal parallel width $\mathtt{W}_2$ and the maximal depth $\mathtt{D}_2$, we generate the scalable CUDA host code as follows:

```
1     // tile time steps
2     for(int t1 = 0; t1 < D2; t1++)
3     {
4        // Kernel Configuration
5
6        // Number of thread blocks in the kernel launch
7        int ND_2_GRIDDim = W2;
8
9        // Number of threads in each thread block corresponds to tile width
10       int ND_2_TBDim = TY;
11
12       //Kernel launch
13       ND_2_Kernel<<< ND_2_GRIDDim, ND_2_TBDim >>>(t1, ga_1, ga_2);
14
15    }
```

Listing B.4: Scaled-up CUDA Host Code.

Adjustments to the Kernel Code

```
1
2 // max width of this DPP (P_2)
3 #define ND_2_W 8
4 // max tile width of this DPP (P_2)
5 #define ND_2_W2 2
6
7 // width of its producer DPP (P_1)
8 #define ND_1_W 9
```

```
 9
10 // Tile width across p-dimension
11 #define TX 4
12 #define TY 4
13 #define ND_2_WTile (TY)
14 #define DPC13_SIZE (1 * ND_2_WTile)
15 //tile offset in x-axis: TX*t1
16 #define tOffsetX (TX * (t1-blockIdx.x))
17 //tile offset in y-axis: TY*p1
18 #define tOffsetY (TY * blockIdx.x)
19
20 // Channel accesses
21 #define ga_2_stride ND_2_W
22 #define sa_2_stride ND_2_WTile
23 #define ga_1_stride ND_1_W
24
25 #define DPC1(t,p) sa_2[ sa_2_stride * ((t-1) - tOffsetX) + ((p) - tOffsetY) ]
26 #define DPC2(t,p) ga_1[ ga_1_stride * (t-p+1) + (p) ]
27 #define DPC3(t,p) sa_2[ sa_2_stride * ((t-1) - tOffsetX) + ((p-1) - tOffsetY)
       ]
28 #define DPC4(t,p) ga_1[ ga_1_stride * (t-p+2) + (p-1) ]
29 #define DPC5(t,p) ga_2[ ga_2_stride * (t) + (p) ]
30
31 __global__ void ND_2_Kernel( int t1, //current tile time step
32                              int *ga_1, //input channel
33                              int *ga_2 //output channel
34                              )
35 {
36     __shared__ int sa_2[DPC13_SIZE];      //self-links
37
38     int in_0;
39     int in_1;
40     int out_0;
41
42     // Mapping: 2-Level CUDA Thread Hierarchy to Processing Entities
43     // blockIdx.x - unique thread block identifier
44   // threadIdx.x - unique thread identified within a thread block
45     // blockDim.x - number of threads in a thread block (block width)
46     int p0 = TY * blockIdx.x + threadIdx.x + 1;
47
48     // Process Control: Execute a number of synchronous time steps in PND tile
49     // Lower and upped bounds ~ the first and the last iteration of the tile
50     for(int t0 = TX * (t1 - blockIdx.x); t0 < TX * (t1 - blockIdx.x) + TX - 1;
           t0++)
51     {
52         ////////////////////////////////////
53         // Process Iteration t0
54         ////////////////////////////////////
55
```

```
56        ////////////////////////////////////
57        // Phase I: READ
58        ////////////////////////////////////
59        if (ND_2IP_1) {
60              in_0 = DPC1(t0, p0);
61        }
62        if (ND_2IP_2) {
63              in_0 = DPC2(t0, p0);
64        }
65        if (ND_2IP_3) {
66              in_1 = DPC3(t0, p0);
67        }
68        if (ND_2IP_4) {
69              in_1 = DPC4(t0, p0);
70        }
71        __syncthreads();
72
73        ////////////////////////////////////
74        // Phase II: EXECUTE
75        ////////////////////////////////////
76        if (ACTIVE_ND_2) {
77              out_0 = predictor(in_0, in_1);
78        }
79
80        ////////////////////////////////////
81        // Phase III: WRITE
82        ////////////////////////////////////
83        if (ND_2OP_1) || (ND_2OP_1_d1)) {
84              DPC13(t0, p0) = out_0;
85        }
86        __syncthreads();
87        if (ND_2OP_1_d2) {
88              DPC5(t0, p0) = out_0;
89        }
90
91     } //end for
92
93
94  } //end ND_2_Kernel
```

Listing B.5: Scaled-up CUDA Kernel Code. Additionally parametrized in grid time step and thread block index.

Appendix C

M-JPEG Encoder

C.1 Overview

Various standards have been developed for compression of digital video signals. The video compression standards can be broadly classified into still image-based compression approaches, and motion estimation-based approaches. Motion JPEG (M-JPEG) belongs to the class of still image-based compression approaches. M-JPEG standard specifies a video codec in which each frame of the video stream is encoded independently as a still image using JPEG standard for image compression. JPEG is a well-known image compression standard, which is named after Joint Photographic Experts Group - the committee that developed and released the standard in 1991.

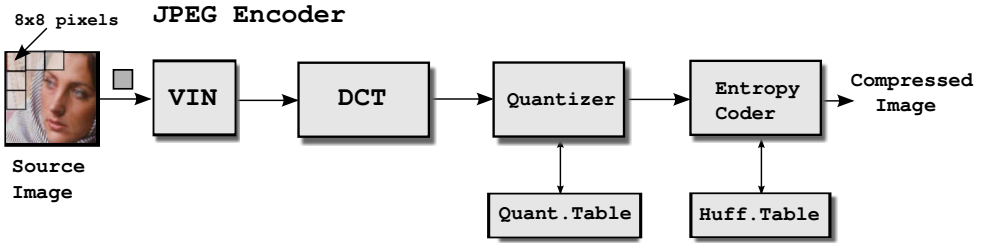


Figure C.1: Block Diagram of JPEG Encoder. In M-JPEG, the still image compression with JPEG encoder is applied to each frame in video sequence individually.

Figure C.1 shows JPEG encoder block diagram. The JPEG encoder partitions the input image into 8×8 blocks of pixels (blocks). Each block is processed indepen-

dently. Each block of the image passes through the DCT module. The DCT module performs a highly efficient 2-dimensional DCT transform to de-correlate the image signal and extract its frequency coefficients. The DCT coefficients are passed to the quantizer, which normalizes the DCT coefficients by a 8×8 quantization matrix and then rounds them off to the nearest integer. The compression ratio, and thus the quality of the encoded image, are determined by the quantization step. The output of the quantization stage is passed to the entropy coder which performs several encoding steps, including run-length encoding and variable-length encoding using the Huffman compression algorithm, on the quantized coefficients. The output of the entropy coder is packed into compressed bitstream to generate the JPEG-encoded image, and stored into a file. The M-JPEG encoder simply applies the JPEG encoding on each frame in the sequence.

As a still image-based compression approach, the M-JPEG does not perform video sequencing (motion) compression, which would allow the encoder to only encode the changes in the video sequence between the frames. However, it has the advantage that the resulting quality of video compression is independent from the motion in the image. As each individual frame is a complete JPEG compressed image, all frames will have the same guaranteed quality, which is not the case with MPEG-based standards. In addition, M-JPEG standard has the smallest latency in image processing, and it is quite easy to understand.

C.2 Code Listings

```
1   TBlock block[VNumBlocks][HNumBlocks];
2
3   for (int t = 0; t < NumFrames; t++) {
4
5   for (int i = 0; i < VNumBlocks; i++)
6     for (int j = 0; j < HNumBlocks; j++)
7      S: mainVIN(&block[i][j]);
8
9   for (int i = 0; i < VNumBlocks; i++)
10    for (int j = 0; j < HNumBlocks; j++)
11     T: mainDCT(block[i][j], &block[i][j]);
12
13  for (int i = 0; i < VNumBlocks; i++)
14    for (int j = 0; j < HNumBlocks; j++)
15     Q: mainQ(block[i][j], &block[i][j]);
16
17  for (int i = 0; i < VNumBlocks; i++)
18    for (int j = 0; j < HNumBlocks; j++)
19     V: mainVLE(block[i][j], &block[i][j]);
20  }
```

Listing C.1: Pseudocode of a M-JPEG Encoder (*main* code snippet)

```
1       int blockIn[8][8];
2       int coeff[8];
3       int tmp[8][8];
4       int blockOut[8][8];
5
6       // Step 1: Pre-shift the pixel values
7     for (i = 0; i < 8; i++)
8       for (j = 0; j < 8; j++)
9        S0: shift(&blockIn[i][j]);
10
11      // Step 2: Perform the first pass of 2D separable integer DCT
12      // Inputs: a whole 8x8 block of pixel values and the coefficients array
13      // Output: a whole 8x8 block of pixel values
14    for (i = 0; i < 8; i++)
15      for (j = 0; j < 8; j++)
16       S1: tmp[i][j] = dotProduct1(blockIn, coeff, i, j);
17
18      // Step 3: Perform the second pass of 2D separable integer DCT
19      // Inputs: a whole 8x8 block of pixel values and the coefficients array
20      // Output: a whole 8x8 block of pixel values
21    for (i = 0; i < 8; i++)
22      for (j = 0; j < 8; j++)
23       S2: blockOut[i][j] = dotProduct2(tmp, coeff, i, j);
24
25      // Step 4: Bound the pixel values to the threshold
26    for (i = 0; i < 8; i++)
27      for (j = 0; j < 8; j++)
28       S3: bound(&blockOut[i][j]);
```

Listing C.2: Code snippet (pseudocode) from the definition of *mainDCT*. The pseudocode illustrates the processing of 8×8 blocks for a single color component.

```
1 void dotProduct1(int blockIn[][8], int coeff[][8], int tmp[][8], int i, int j)
2 {
3   int sum = 0;
4   for (int k = 0; k < 8; k++)
5     {
6      sum += blockIn[i][k] * (coeff[j][k]>>16);
7      tmp[i][j] = sum >> 8;
8   }
9 }
10
11 void dotProduct2(int tmp[][8], int coeff[][8], int blockOut[][8], int i, int j
     )
```

```
12 {
13    int sum = 0;
14    for (k = 0; k < 8; k++)
15     {
16       sum += (coeff[i][k]>>16) * tmp[k][j];
17       blockOut[i][j] = sum >> 8;
18    }
19 }
```

Listing C.3: Code snippet (pseudocode) from the definition of DCT passes.

C.3 M-JPEG PPN

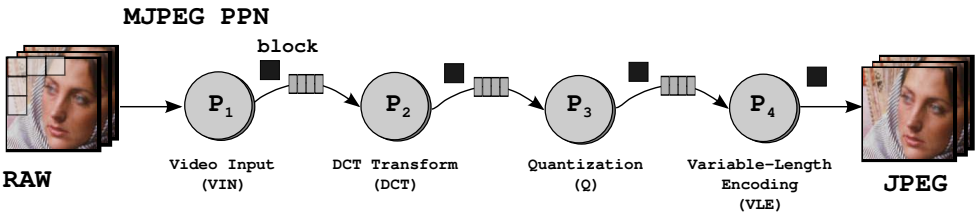


Figure C.2: Automatically generated M-JPEG PPN is a pipeline composed of four major tasks. The input to the pipeline is a stream of frames in raw format, the M-JPEG PPN performs JPEG image compression on each frame individually.

The M-JPEG SANLP in Listing C.1 is transformed by the COMPAAN compiler into four tasks represented by PPN processes, as illustrated in Figure C.2. Each PPN process executes one statement of the SANLP and its enclosing loop nest. For example, the `DCT` node (P_2) executes the statement `T : mainDCT(block[i][j], &block[i][j])` on the iteration domain resulting from for loops f, i, and j that enclose the statement T. The four processes form a straight-forward processing pipeline. The processes are connected via channels implemented as FIFO buffers and exchange data via tokens. The token data type in the PPN generated by the COMPAAN compiler always equals the data type of the variables in the source code, which is here a *block* of the picture that contains 8×8 pixels. Following the default PPN implementation and mapping approach in Compaan, PPN processes are assigned as tasks for execution to different threads. Each PPN process task *sequentially* executes iterations of the for loops encapsulating its program statement (e.g. the `DCT` node executes its copy of the code on lines 3, 9, 10, 11 in Listing C.1). This parallelization approach is used for task and pipeline parallel execution on multicore platforms.