Parallelized AES-256 Encryption Using CUDA

Hao-Chien Lin
UW Electrical and Computer
Engineering

***Abstract* — Two parallel methods are implemented in this project to perform AES-256. One method is called "Parallelization by State Arrays" and the other method is called "Parallelization by Dynamic Parallelism". Each method has its strengths. However, the former method has proved to be more efficient overall.**

I. OVERVIEW

AES encryption is a widely used encryption method. There are 128-bit, 192-bit, 256-bit AES encryption available, with 256-bit AES encryption being the most secure. The illustration of the procedures required to perform AES-256 is shown in Fig. 1.

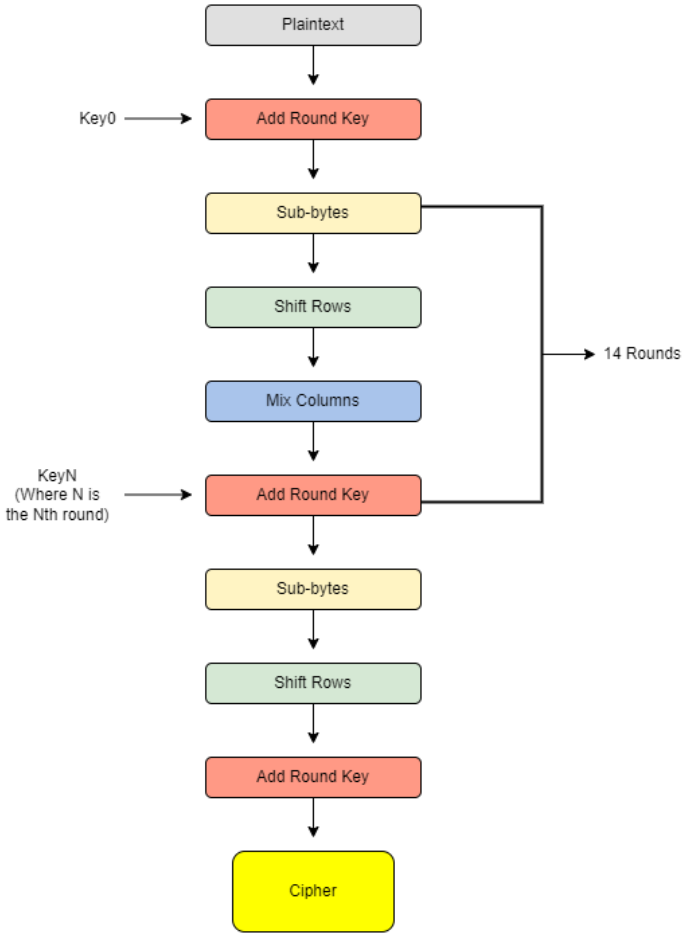

Fig. 1. AES-256 Procedures

A. *Key Expansion*

A total of 15 keys are required to complete the encryption. Therefore, key expansion needs to be done prior to encryption. Each key is 16 bytes in length, meaning that we need to expand the 32 bytes of input key to 15×16 = 240 bytes. The procedure to perform key expansion is illustrated in Fig. 2.

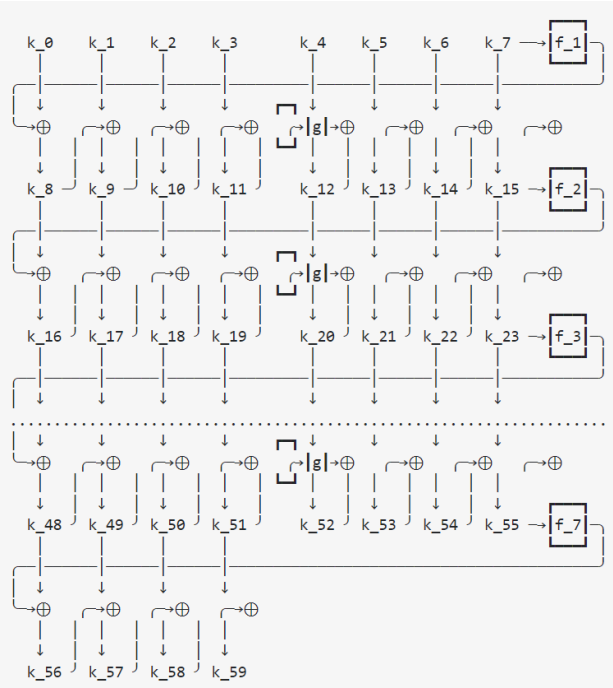

Fig. 2. Key Expansion Procedures

Key expansion in this project is done sequentially. Each k in the illustration represents a 4-byte word. A single key can be divided into 4 words. The total number of words needed would be 240 bytes / 4bytes = 60. The input key of 32-bytes occupies k_0 to k_7.

The g function is done by performing a s-box lookup of each element in the input word. The resulting lookup value of each element would become the value of the element itself. For instance, if an input word consists of the hexadecimal numbers [12, 43, F3, AE], the output of the g function would be [FD, 18, 75, 87].

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Fig. 3. S-box

The f function is done by performing the steps “rotation word”, “sub-word”, and “R-Con”.

- *Rotation word:* Perform a one-byte left circular shift. For example:

$$RotWord([b_0\ b_1\ b_2\ b_3]) = [b_1\ b_2\ b_3\ b_0]$$

- *Sub-word:* Perform a s-box lookup of each element of the word. This step is the exact same as the g function.

- *R-Con:* Perform a XOR between an element in the R-Con matrix and the first element of the input word. The element of the R-Con matrix to perform XOR depends on the number of f functions already performed. For instance, if the current f function is the second f function that was performed, then the second element in the R-Con matrix is the element to perform XOR.

8d	01	02	04	08	10	20	40	80	1b	36	6c	d8	ab	4d	9a
2f	5e	bc	63	c6	97	35	6a	d4	b3	7d	fa	ef	c5	91	39
72	e4	d3	bd	61	c2	9f	25	4a	94	33	66	cc	83	1d	3a
74	e8	cb	8d	01	02	04	08	10	20	40	80	1b	36	6c	d8
ab	4d	9a	2f	5e	bc	63	c6	97	35	6a	d4	b3	7d	fa	ef
c5	91	39	72	e4	d3	bd	61	c2	9f	25	4a	94	33	66	cc
83	1d	3a	74	e8	cb	8d	01	02	04	08	10	20	40	80	1b
36	6c	d8	ab	4d	9a	2f	5e	bc	63	c6	97	35	6a	d4	b3
7d	fa	ef	c5	91	39	72	e4	d3	bd	61	c2	9f	25	4a	94
33	66	cc	83	1d	3a	74	e8	cb	8d	01	02	04	08	10	20
40	80	1b	36	6c	d8	ab	4d	9a	2f	5e	bc	63	c6	97	35
6a	d4	b3	7d	fa	ef	c5	91	39	72	e4	d3	bd	61	c2	9f
25	4a	94	33	66	cc	83	1d	3a	74	e8	cb	8d	01	02	04
08	10	20	40	80	1b	36	6c	d8	ab	4d	9a	2f	5e	bc	63
c6	97	35	6a	d4	b3	7d	fa	ef	c5	91	39	72	e4	d3	bd
61	c2	9f	25	4a	94	33	66	cc	83	1d	3a	74	e8	cb	8d

Fig. 4. R-con matrix

Fig. 5.

B. Encrytion Steps

As seen in figure 1, AES-256 encryption consists of 4 distinct steps. The steps are: Add Round Key, Sub-bytes, Shift Rows, and Mix Columns.

Add Round Key: Perform a XOR between each element of the state array and the current key under consideration.

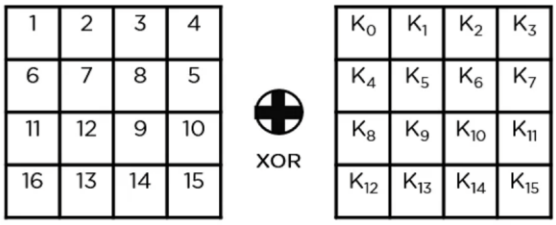

Fig. 6. Add Round Key

A state array is an array of 16 hexadecimal 8-bit unsigned integers. Calulations amongst steps are done in units of these 16-byte state arrays. The initial state array is derived from plaintext, having characters represented as 1-byte hexadecimal unisgned integers. Therefore, the input plaintext can be divided into a certain number of 16-byte state arrays.

Sub-bytes: Perform a s-box lookup of each element in the state array. The resulting lookup value will become the value of the element itself.

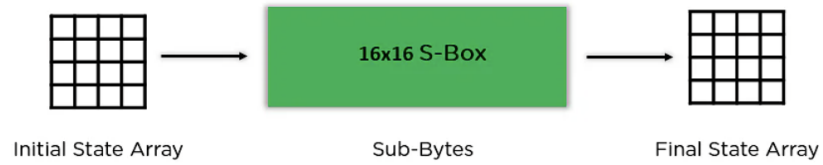

Fig. 7. Sub-Bytes

Shift Rows: Left shift the 0th, 1st, 2nd , and 3rd rows of the state array by 0-byte, 1-byte, 2-bytes, and 3-bytes respectively.

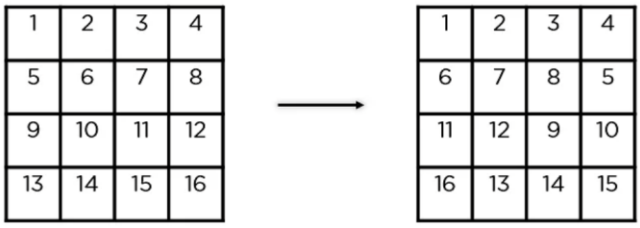

Fig. 8. Shift Rows

Mix Columns: Perform a matrix multiplication in Galois Fields between the mix column matrix and state array. The mix column matrix consists of set values as seen on the left grid in Fig. 8.

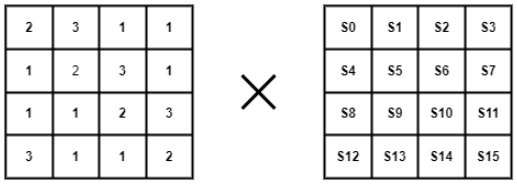

Fig. 9. Mix Columns

Multiplication is done in Galois Fields, which differs from regular multiplication. Multiplication in mix columns can be generalized into 3 cases.

Multiply by 1: Results in the original multiplicand.

Multiply by 2: Left shift the multiplicand by one bit. If the 8th bit of the original multiplicand before the shift was 1, then XOR the multiplicand with 27. Otherwise, if the 8th bit of the original multiplicand was not 1, do not perform the XOR.

Multiply by 3: Multiply the multiplicand by 2 just like the above instructions, then XOR the original multiplicand with the result from multiplying the multiplicand by 2.

When multiplication is done between one row in the mix matrix and one column in the state array, XOR the results just like how multiplication results are added in regular matrix multiplication.

C. *Mode of Encryption*

AES encryption is a block cipher which performs encryption on 16-byte state arrays. Since the input plaintext may not always be multiples of 16-bytes, a mechanism is needed to pad the state arrays with characters if it

contains less than 16-bytes of data. Also, a mechanism is needed to combine each 16-byte state array into a single resulting cipher. The mechanism used to perform these operations is ECB mode.

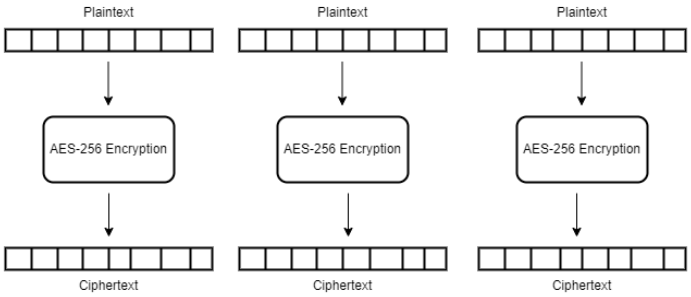

Fig. 10. ECB Mode

Padding is required to pad state arrays to ensure each state array contains 16-bytes of data. The method of padding used was to pad with bytes equal to the value of the number of padding bytes required. For instance, if the state array that is needed for padding is [66, 6F, 72], 5 hexadecimal values with the value 05 would be used to pad this state array. The resulting state array would become [66, 6F, 72, 05, 05, 05, 05, 05].

II. CONCEPT/SEMANTIC AND IMPLEMENTATION OF PARALLELIZATION

A. Parallelization by State Arrays

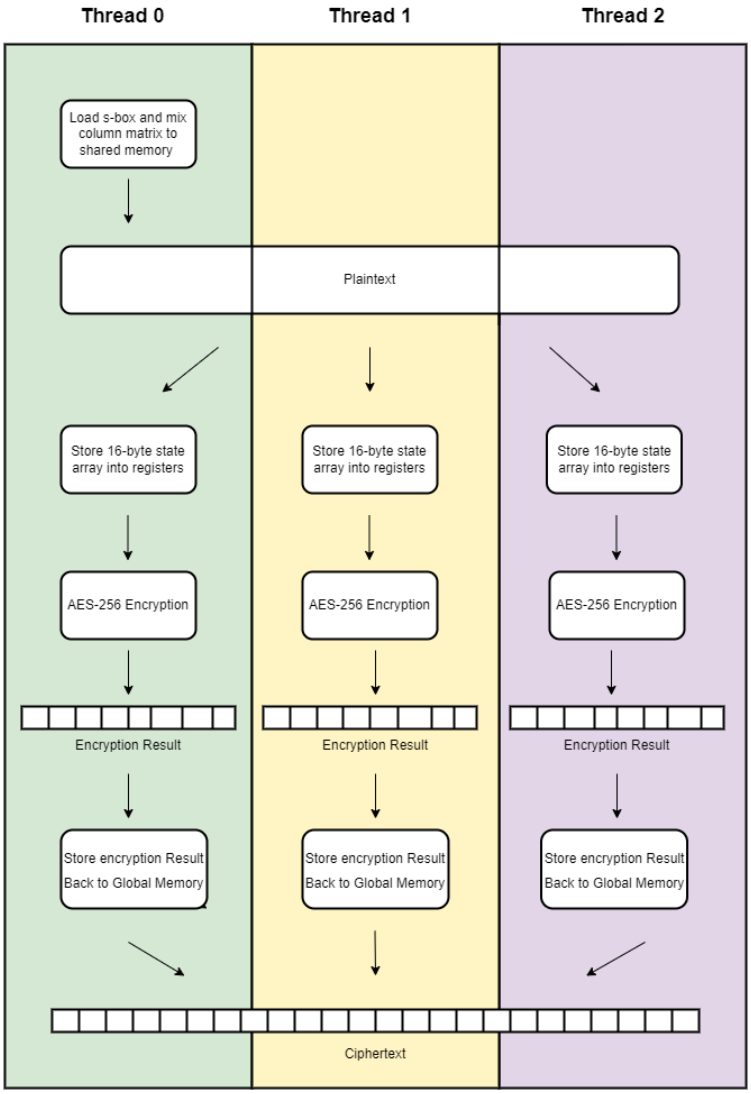

Fig. 11. Illustration of Parallization by State Arrays

The number of threads that will be allocated to the kernel is the number of state arrays that can be partitioned from the plaintext. Therefore, this method is great for plaintext that are very long. The larger the input plaintext, the greater the amount of parallelization.

The s-box and the mix column matrix, which are used extensively throughout the AES-256 encryption process in the Sub-bytes and Mix Columns steps are first stored into shared memory for faster data access.

The state array, which will be used in essentially every step of the encryption process is stored in the register. This allows for fast read and write. Also, since only 16 bytes of data are stored, and the Turing T4 allows a maximum of 255 32-bit registers per thread, it is safe to say that register spilling will not occur.

The sub-bytes, shift rows, and mix columns steps are all performed sequentially. Dynamic parallelism cannot be utilized to perform these steps in parallel because data that these steps need, such as s-box, mix column matrix, and state arrays are stored in shared memory and registers. Data that are stored in shared memory and registers cannot be passed into child kernels. Therefore, the “Parallelization by Dynamic

Parallelism" method was implemented in which the s-box, mix column matrix, and state arrays are stored in global memory. This way, the sub-bytes , shift rows, and mix columns steps can be implemented in parallel with child kernels.

B. *Parallelization by Dynamic Parallelism*

This method also breaks the plaintext into state arrays of 16 bytes. Each thread is then responsible of each state array. However, the difference is that each of the Add Round Key, Sub-bytes, Shift Rows, and Mix Columns steps are launched as child kernels and the state array, s-box, and mix matrix are stored in global memory. By launching these steps as child kernels, these steps can be parallelized. In the "Parallelization by State Arrays" method, the Add Round Key, Sub-bytes, Shift Rows, and Mix Columns steps were all implemented sequentially. Some down sides of this method of implementation are that the s-box and mix matrix cannot be stored in shared memory because they must be passed to child kernels. Also, the state array cannot be stored in registers because they must be passed to child kernels too. Moreover, the amount of parallelism that can be achieved with child kernels are minimal considering that each child kernel only performs at most 16 operations. Only 16 threads are allocated to each child kernel for parallelization.

```cuda
__global__ void AESEncryption(uint8_t* text, uint8_t* key, uint8_t* sBox, uint8_t* mMatrix, int blockNumber) {

    int thread = threadIdx.x;
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(4, 4, 1);
    if (thread < blockNumber) {

        addRoundKey << <1, 16 >> > (text, key, thread, 0);
        cudaDeviceSynchronize();

        for (int i = 1; i < 14; i++) {
            subBytes << <1, 16 >> > (text, sBox, thread);
            cudaDeviceSynchronize();
            shiftRows << <1, 16 >> > (text, thread);
            cudaDeviceSynchronize();
            mixColumns << <dimGrid, dimBlock >> > (text, mMatrix, thread);
            cudaDeviceSynchronize();
            addRoundKey << <1, 16 >> > (text, key, thread, i);
            cudaDeviceSynchronize();
        }
        subBytes << <1, 16 >> > (text, sBox, thread);
        cudaDeviceSynchronize();
        shiftRows << <1, 16 >> > (text, thread);
        cudaDeviceSynchronize();
        addRoundKey << <1, 16 >> > (text, key, thread, 14);
        cudaDeviceSynchronize();

    }
}
```

```cuda
__global__ void AESEncryption(uint8_t* text, uint8_t* key, int blockNumber) {

    int thread = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ uint8_t sBox[256];
    __shared__ uint8_t mixMatrix[16];

    if (thread < blockNumber) {

        if (thread == 0) {
            load(sBox, mixMatrix);
        }
        __syncthreads();

        uint8_t currBlock[16];
        for (int i = 0; i < 16; i++) {
            currBlock[i] = text[thread * 16 + i];
        }
        addRoundKey(currBlock, key, 0);

        for (int i = 1; i < 14; i++) {
            subBytes(currBlock, sBox);
            shiftRows(currBlock);
            mixColumns(currBlock, mixMatrix);
            addRoundKey(currBlock, key, i);
        }
        subBytes(currBlock, sBox);
        shiftRows(currBlock);
        addRoundKey(currBlock, key, 14);


        for (int i = 0; i < 16; i++) {
            text[thread * 16 + i] = currBlock[i];
        }
    }
}
```

Fig. 12. Main kernel of Parallelization by Dynamic (Left) vs Parallelization by State Arrays (right). It can be observed that each step in the Parallelization by Dynamic method is launched as a child kernel.

```cuda
__global__ void addRoundKey(uint8_t* text, uint8_t* key, int index, int round) {

    int thread = threadIdx.x;

    if (thread < 16) {
        text[index*16 + thread] ^= key[round * 16 + thread];
    }

}
```

Fig. 13. Parallization of the Add Round Key step

```cuda
__global__ void subBytes(uint8_t* text, uint8_t* sBox, int index) {

    int thread = threadIdx.x;

    if (thread < 16) {
        text[index * 16 + thread] = sBox[text[index * 16 + thread]];
    }

}
```

Fig. 14. Parallization of the Sub-bytes step

```
__global__ void shiftRows(uint8_t* text, int index) {

    int thread = threadIdx.x;
    uint8_t out[16];
    if (thread < 16 && thread >= 4) {

        switch (thread) {
        case 4:
            out[4] = text[index * 16 + 5];
            break;
        case 5:
            out[5] = text[index * 16 + 6];
            break;
        case 6:
            out[6] = text[index * 16 + 7];
            break;
        case 7:
            out[7] = text[index * 16 + 4];
            break;
        case 8:
            out[8] = text[index * 16 + 10];
            break;
        case 9:
            out[9] = text[index * 16 + 11];
            break;
        case 10:
            out[10] = text[index * 16 + 8];
            break;
        case 11:
            out[11] = text[index * 16 + 9];
            break;
        case 12:
            out[12] = text[index * 16 + 15];
            break;
        case 13:
            out[13] = text[index * 16 + 12];
            break;
        case 14:
            out[14] = text[index * 16 + 13];
            break;
        case 15:
            out[15] = text[index * 16 + 14];
            break;
        }

        __syncthreads();

        text[index * 16 + thread] = out[thread];
    }
}
```

Fig. 15. Parallization of the Shift Rows step

```
__global__ void mixColumns(uint8_t* text, uint8_t* mixMatrix, int index) {

    __shared__ uint8_t ds_mix[TILE_WIDTH][TILE_WIDTH];
    __shared__ uint8_t ds_state[TILE_WIDTH][TILE_WIDTH];

    int width = 4;
    uint8_t ty = threadIdx.y;
    uint8_t tx = threadIdx.x;
    uint8_t row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    uint8_t col = blockIdx.x * TILE_WIDTH + threadIdx.x;
    uint8_t mixResult[16];
    uint8_t pval = 0;
    uint8_t tempResult = 0;

    for (int ph = 0; ph < ceil(width / (float)TILE_WIDTH); ++ph) {

        if ((row < width) && (ph * TILE_WIDTH + tx) < width)
        {
            ds_mix[ty][tx] = mixMatrix[row * width + ph * TILE_WIDTH + tx];
        }
        else
        {
            ds_mix[ty][tx] = 0;
        }
        if ((ph * TILE_WIDTH + ty) < width && col < width)
        {
            ds_state[ty][tx] = text[((ph * TILE_WIDTH + ty) * width + col) + index*16];
        }
        else
        {
            ds_state[ty][tx] = 0;
        }

        __syncthreads();

        if (row < width && col < width)
        {
            for (int i = 0; i < TILE_WIDTH; ++i)
            {
                uint8_t currState = ds_state[i][tx];

                switch (ds_mix[ty][i]) {
                case 1:
                    tempResult = ds_state[i][tx];
                    break;
                case 2:
                    if (currState >= 128) {
                        tempResult = currState << 1;
                        tempResult ^= 27;
                    }
                    else {
                        tempResult = currState << 1;
                    }
                    break;
                case 3:
                    if (currState >= 128) {
                        tempResult = currState << 1;
```

Fig. 16. Part of the parallization of the Mix Columns step

The parallization of these child kernels all involve allocating one thread to perfom a single parallelizable task. Mix Columns is the most sophisticated out of these kernels as it utilizes tilied matrix multiplication. However, it differs from the classic tiled matrix multiplication since multiplication is done in Galois Fields.

III. IDENTIFY PARALLEL PATTERNS USED

A. *Parallelization by State Arrays*

- Encryption is parallelized by dividing up the input plaintext into blocks of 16-byte state arrays. Each thread is responsible of encrypting one state array. At the end, all state arrays would merge into one cipher.
- The s-box and mix matrix were stored in shared memory for fast data read.
- The state arrays were stored in the registers of each thread to ensure the fastest read and write. State arrays are data that are used most extensively throughout the encryption process; thus, it is vital that it is stored in fast memory.

B. *Parallelization by Dynamic Parallelism.*

- Encryption is parallelized by dividing up the input plaintext into blocks of 16-byte state arrays. Each thread is responsible of encrypting one state array.
- Add Round Key: A child kernel was launched with 16 threads to perform XORs between the state array and the key in parallel.
- Sub-Bytes: A child kernel was launched with 16 threads to perform s-box data lookup in parallel.
- Shift Rows: A child kernel was launched with 16 threads to shift elements in the state array in parallel.

- Memory Coalescing: Access to global memory for all child kernels are in the form "A[(expression with terms independent of threadIdx.x) + threadIdx.x]. Thus, memory access is coalesced.
- Mix Columns. A child kernel was launched with 16 thread to perform tiled matrix multiplication between the state array and the mix matrix. Some changes were made to the tiled matrix multiplication to accommodate for Galois Fields.

IV. Source Code for Host & Kernel

- *Parallelization_by_State_Arrays.cu*: Host and Kernel code for Parallelization by State Arrays.
- *Parallelization_by_Dynamic.cu*: Host and Kernel code for Parallelization by Dynamic Parallelism.
- *Sequential.cu*: Code for sequential execution.
- *plaintext.txt*: File used to store the input plaintext.
- *key.txt*: File used to store the input key.
- *cipher.txt*: File used to store the AES-256 encryption result.
- *plaintext1-12000*: Dataset with varying sizes of plaintext. The number in the name is the size of the plaintext in kilobytes.

V. Profiling Reports, Discussion, and Analysis

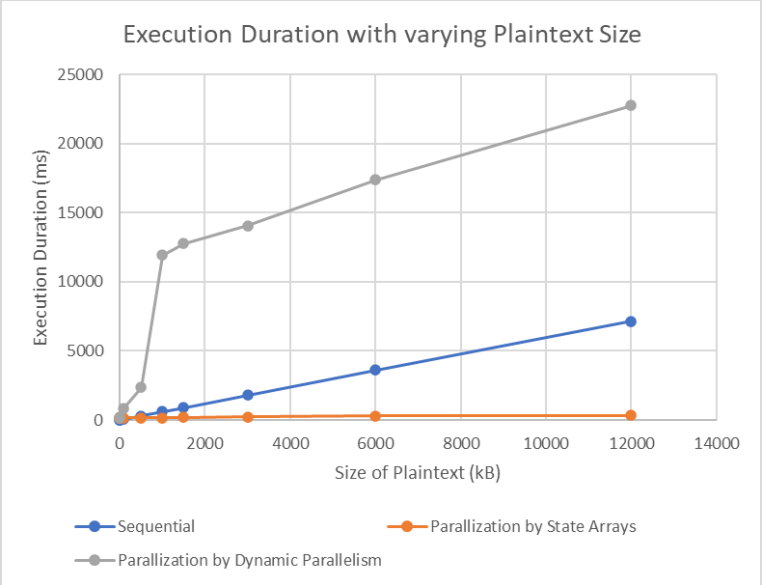

Fig. 17. Execution duration of each method with varying plaintext size

The execution duration of the 2 parallelization methods and the sequential algorithm was calculated using the windows high performance timer API. The best performing method is the Parallelization by State Array method. While the worse performing method was the Parallelization by Dynamic method. The execution time of the sequential algorithm increased linearly with input size. For the Parallelization by Dynamic method, the execution time initially increased much faster than the sequential method. However, for plaintext with sizes greater than 1000kB, the rate of increase of its execution duration became less than the sequential method. Therefore, when plaintext size is very large, there is a chance that the Parallelization by Dynamic method can be faster than the sequential method.

From the result of the timing, it can be safe to say that parallelizing by storing the state array in registers, and the s-box and mix matrix in shared memory is the fastest method. Even through the individual steps of the Parallelization by Dynamic method are parallelized, the parallelization only involves 16 threads. This

improvement is insignificant compared to the delay caused by data access through global memory in most of data loading done in the Parallelization by Dynamic method.

All the analysis from this point below are performed using a plaintext dataset of 3000kB.

The roofline graph does not show the achieved roofline as seen in Fig. 18. However, it does show that the peak performance boundary and memory bandwidth boundary are greater for the Parallelization by State Array method. Therefore, I am unable to discern whether these algorithms are compute or memory bound.

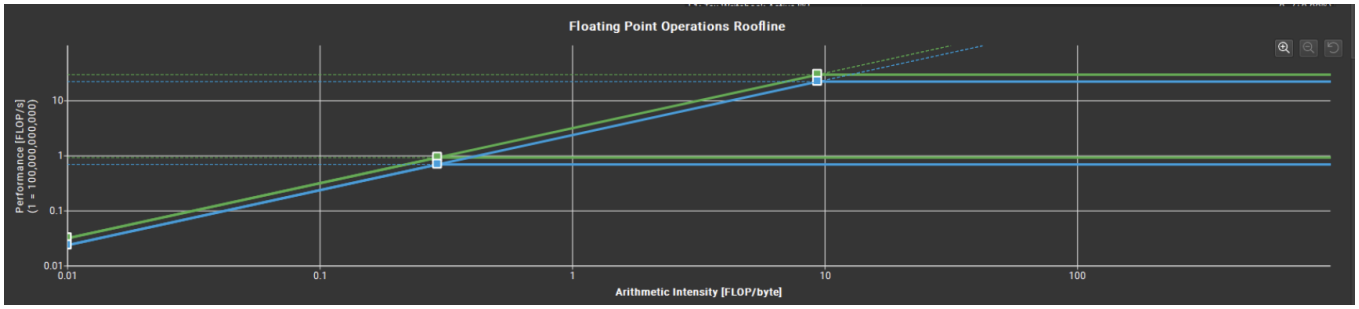

Fig. 18. Roofline graph of Parallelization by State Array (green) and Parallelization by Dynamic (Blue)

It can be observed in Fig. 19 that the registers per thread value of the Parallelization by State Array method is a lot higher than the Parallelization by Dynamic method. This coincides with the implementation in which more data is stored in registers in the Parallelization by State Array method. Data such as the state array is solely stored in registers in the method.

Also, it can be observed in Fig. 19 that the time it takes to execute the Parallelization by State Array method is magnitudes of times faster than the Parallelization by Dynamic method.

	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
P_By_State_Arrays	Parallelization_by_State_Arrays	523 - AESEncryption (6000, 1, 1)x(32, 1, 1)	1.26 msecond	735,875	58	0 - Tesla T4	583.40 cycle/usecond	7.5	[15348] Parallelization_by_State_Arrays.exe
P_By_Dynamic	Parallelization_by_Dynamic	574 - AESEncryption (6000, 1, 1)x(32, 1, 1)	48.77 second	21,291,218,200	32	0 - Tesla T4	436.58 cycle/usecond	7.5	[392] kernel.exe

Fig. 19. Summary of metrics of the Parallelization by State Array and Parallelization by Dynamic methods

A. Parallelization by State Arrays

The main kernel was launched with 6000 blocks of 32 threads as seen in Fig. 20. The size of the blocks is determined dynamically according to the input plaintext size.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	6,000 (+0.00%)	Registers Per Thread [register/thread]	58 (+0.00%)
Block Size	32 (+0.00%)	Static Shared Memory Per Block [byte/block]	272 (+0.00%)
Threads [thread]	192,000 (+0.00%)	Dynamic Shared Memory Per Block [byte/block]	0 (+0.00%)
Waves Per SM	9.38 (+0.00%)	Driver Shared Memory Per Block [byte/block]	0 (+0.00%)
Function Cache Configuration	cudaFuncCachePreferNone (cudaFuncCachePreferNone)	Shared Memory Configuration Size [Kbyte]	32.77 (+0.00%)

Fig. 20. Launch Statistics

The peak memory and peak compute performance is shown in Fig. 21. Its values are higher compared to the Parallelization by Dynamic method.

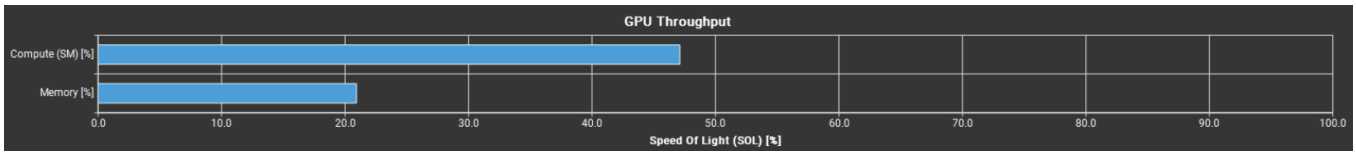

Fig. 21. Speed of Light

The chart in Fig. 22 shows a fair amount of data being read and written to shared memory. This reflects the reality considering how the s-box and the mix column matrix are stored in shared memory. Also, it can be seen that there are no interactions with local memory. This shows that register spilling did not occur when storing state arrays in registers. The hit rate of the L1/TEX memory is very high, standing at 97.85%, this is a contribution factor to the faster execution time compared to the Parallelization by Dynamic method which has a lower L1/TEX hit rate.

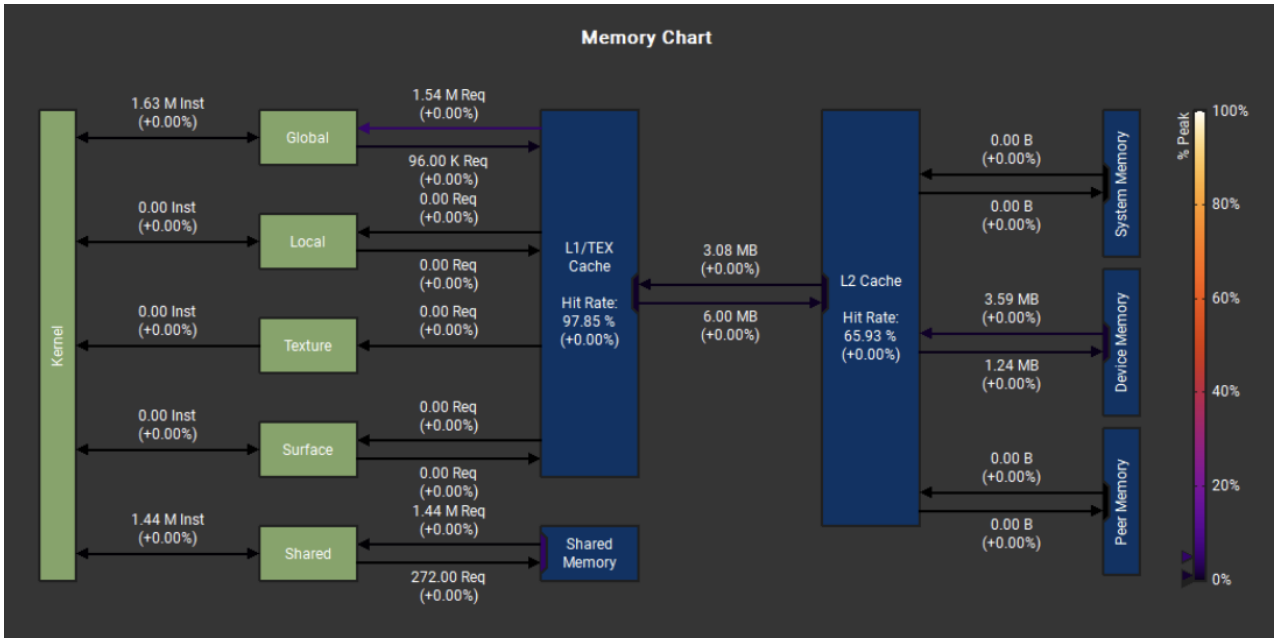

Fig. 22. Memory Chart for Parallelization by State Arrays

As shown in Fig. 23, in order to avoid access to global memory for storing s-box and mix column matrix values, the values were explicited assigned to shared memory.

```
sBox[0] = 0x63; sBox[1] = 0x7c; sBox[2] = 0x77; sBox[3] = 0x7b; sBox[4]
sBox[16] = 0xca; sBox[17] = 0x82; sBox[18] = 0xc9; sBox[19] = 0x7d; sBo
sBox[32] = 0xb7; sBox[33] = 0xfd; sBox[34] = 0x93; sBox[35] = 0x26; sBo
sBox[48] = 0x4; sBox[49] = 0xc7; sBox[50] = 0x23; sBox[51] = 0xc3; sBox
sBox[64] = 0x9; sBox[65] = 0x83; sBox[66] = 0x2c; sBox[67] = 0x1a; sBox
sBox[80] = 0x53; sBox[81] = 0xd1; sBox[82] = 0x0; sBox[83] = 0xed; sBox
sBox[96] = 0xd0; sBox[97] = 0xef; sBox[98] = 0xaa; sBox[99] = 0xfb; sBo
sBox[112] = 0x51; sBox[113] = 0xa3; sBox[114] = 0x40; sBox[115] = 0x8f;
sBox[128] = 0xcd; sBox[129] = 0xc; sBox[130] = 0x13; sBox[131] = 0xec;
sBox[144] = 0x60; sBox[145] = 0x81; sBox[146] = 0x4f; sBox[147] = 0xdc;
sBox[160] = 0xe0; sBox[161] = 0x32; sBox[162] = 0x3a; sBox[163] = 0xa;
sBox[176] = 0xe7; sBox[177] = 0xc8; sBox[178] = 0x37; sBox[179] = 0x6d;
sBox[192] = 0xba; sBox[193] = 0x78; sBox[194] = 0x25; sBox[195] = 0x2e;
sBox[208] = 0x70; sBox[209] = 0x3e; sBox[210] = 0xb5; sBox[211] = 0x66;
sBox[224] = 0xe1; sBox[225] = 0xf8; sBox[226] = 0x98; sBox[227] = 0x11;
sBox[240] = 0x8c; sBox[241] = 0xa1; sBox[242] = 0x89; sBox[243] = 0xd;
```

Fig. 23. Storing s-box values in shared memory

No memory coalescing patterns were utilized in the Parallelization by State Arrays method. As a result, the sectors/request rate, which is 1.94, as shown in Fig. 24 is not ideal. The ideal sector/request rate would be 4.

L1/TEX Cache

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM	% Peak to SM
Local Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)				
Global Load	1,536,000 (+0.00%)	1,536,000 (+0.00%)	1,536,000 (+0.00%)	5.22 (+0.00%)	2,975,792 (+0.00%)	1.94 (+0.00%)	96.76 (+0.00%)	95,225,344 (+0.00%)	96,307 (+0.00%)	0.33 (+0.00%)	1,537,240 (+0.00%)	5.22 (+0.00
Surface Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00
Texture Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)		
Global Store	96,000 (+0.00%)	96,000 (+0.00%)	96,000 (+0.00%)	0.33 (+0.00%)	1,535,792 (+0.00%)	16.00 (+0.00%)	99.94 (+0.00%)	49,145,344 (+0.00%)	187,499 (+0.00%)	0.64 (+0.00%)	-	
Local Store	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)			-	
Surface Store	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	-	
Global Reduction	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	-	
Surface Reduction	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	-	
Global Atomic ALU	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	see above	see abc
Global Atomic CAS	0 (+0.00%)											
Surface Atomic ALU	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	see above	see abc
Surface Atomic CAS	0 (+0.00%)											
Loads	1,536,000 (+0.00%)	1,536,000 (+0.00%)	1,536,000 (+0.00%)	5.22 (+0.00%)	2,975,792 (+0.00%)	1.94 (+0.00%)	96.76 (+0.00%)	95,225,344 (+0.00%)	96,307 (+0.00%)	0.33 (+0.00%)	1,537,240 (+0.00%)	5.22 (+0.00
							16.00	99.94				

Fig. 24. L1/TEX Cache Information

It is interesting to see in Fig. 25 that the sectors/request rate for the L2 cache is 3.96, which is very close to 4. This means that the memory access to the L2 cache is done in a more efficient way compared to access to the L1 cache.

L2 Cache

	Requests	Sectors	Sectors/Req	% Peak	Hit Rate	Bytes	Throughput	Sector Misses to Device	Sector Misses to System	Sector Misses to Peer
L1/TEX Load	24,317 (+0.00%)	96,307 (+0.00%)	3.96 (+0.00%)	0.28 (+0.00%)	0.33 (+0.00%)	3,081,824 (+0.00%)	2,443,285,891.87 (+0.00%)	95,996 (+0.00%)	0 (+0.00%)	0 (+0.00%)
L1/TEX Store	46,787 (+0.00%)	183,923 (+0.00%)	3.93 (+0.00%)	0.53 (+0.00%)	100 (+0.00%)	5,885,536 (+0.00%)	4,666,083,162.09 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
L1/TEX Atomic ALU	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
L1/TEX Atomic CAS	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
L1/TEX Reduction	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
L1/TEX Total	70,265 (+0.00%)	283,806 (+0.00%)	4.04 (+0.00%)	0.82 (+0.00%)	66.12 (+0.00%)	9,081,792 (+0.00%)	7,200,091,331.15 (+0.00%)	95,996 (+0.00%)	0 (+0.00%)	0 (+0.00%)
ECC Total	-	8 (+0.00%)	-	0.00 (+0.00%)	-	256 (+0.00%)	202,958.11 (+0.00%)	8 (+0.00%)	-	-
GPU Total	71,874 (+0.00%)	284,843 (+0.00%)	3.96 (+0.00%)	0.83 (+0.00%)	66.17 (+0.00%)	9,114,976 (+0.00%)	7,226,399,776.75 (+0.00%)	96,008 (+0.00%)	0 (+0.00%)	0 (+0.00%)

Fig. 25. L2 Cache Information

Minimal amount of control divergence exists in the code. The only control divergence exists in the main kernel. Threads in the warp would have to wait for thread 0 to finish loading data into shared memory before they can start performing their tasks as shown in Fig. 26.

```
__global__ void AESEncryption(uint8_t* text, uint8_t* key, int blockNumber) {

    int thread = blockDim.x * blockIdx.x + threadIdx.x;

    __shared__ uint8_t sBox[256];
    __shared__ uint8_t mixMatrix[16];

    if (thread < blockNumber) {

        if (thread == 0) {
            load(sBox, mixMatrix);
        }
        __syncthreads();

        uint8_t currBlock[16];
        for (int i = 0; i < 16; i++) {
            currBlock[i] = text[thread * 16 + i];
        }
        addRoundKey(currBlock, key, 0);

        for (int i = 1; i < 14; i++) {
            subBytes(currBlock, sBox);
            shiftRows(currBlock);
            mixColumns(currBlock, mixMatrix);
            addRoundKey(currBlock, key, i);
        }
        subBytes(currBlock, sBox);
        shiftRows(currBlock);
        addRoundKey(currBlock, key, 14);


        for (int i = 0; i < 16; i++) {
            text[thread * 16 + i] = currBlock[i];
        }
    }
}
```

Fig. 26. Control divergence exists when thread 0 needs to load data into shared memory

As shown in Fig. 27, the theoretical occupancy is 50%. It is limited by the number of blocks that can fit on the SM. The achieved occupancy is 47.95%, which is very close to the theoretical value. An occupancy of 47.95% can help cover some latency during global loads.

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50 (+0.00%)	Block Limit Registers [block]	32 (+0.00%)
Theoretical Active Warps per SM [warp]	16 (+0.00%)	Block Limit Shared Mem [block]	64 (+0.00%)
Achieved Occupancy [%]	47.95 (+0.00%)	Block Limit Warps [block]	32 (+0.00%)
Achieved Active Warps Per SM [warp]	15.34 (+0.00%)	Block Limit SM [block]	16 (+0.00%)

Fig. 27. Occupancy Information

B. *Parallelization by Dynamic Parallelism*

The main kernel was launched with 6000 blocks of 32 threads as seen in Fig. 28. The size of the blocks is determined dynamically according to the input plaintext size.

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	6,000 (+0.00%)	Registers Per Thread [register/thread]	32 (-44.83%)
Block Size	32 (+0.00%)	Static Shared Memory Per Block [byte/block]	0 (-100.00%)
Threads [thread]	192,000 (+0.00%)	Dynamic Shared Memory Per Block [byte/block]	0 (+0.00%)
Waves Per SM	9.38 (+0.00%)	Driver Shared Memory Per Block [byte/block]	0 (+0.00%)
Function Cache Configuration	cudaFuncCachePreferNone (cudaFuncCachePreferNone)	Shared Memory Configuration Size [Kbyte]	32.77 (+0.00%)

Fig. 28. Launch Statistics

The peak memory and peak compute performance is very low compared to the Parallelization by Static Arrays method as seen in Fig. 29.

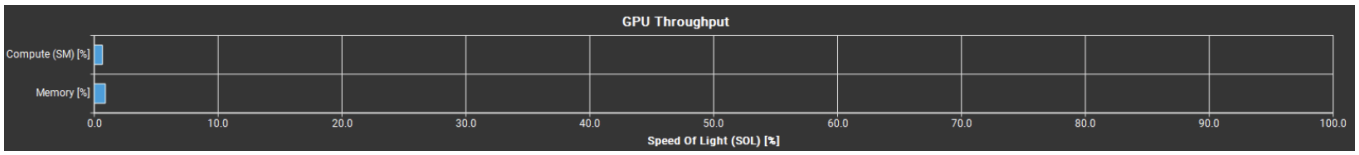

Fig. 29. Speed of Light

Some data is read and stored in the shared memory as seen in Fig. 30. This result is as anticipated because the Mix Columns step utilizes tiled matrix multiplication. Tiles are stored in shared memory. The hit rate for L1/TEX cache is lower compared to the hit rate of the Parallelization by State Arrays method. This is another contribution factor of the slower execution time of this method.

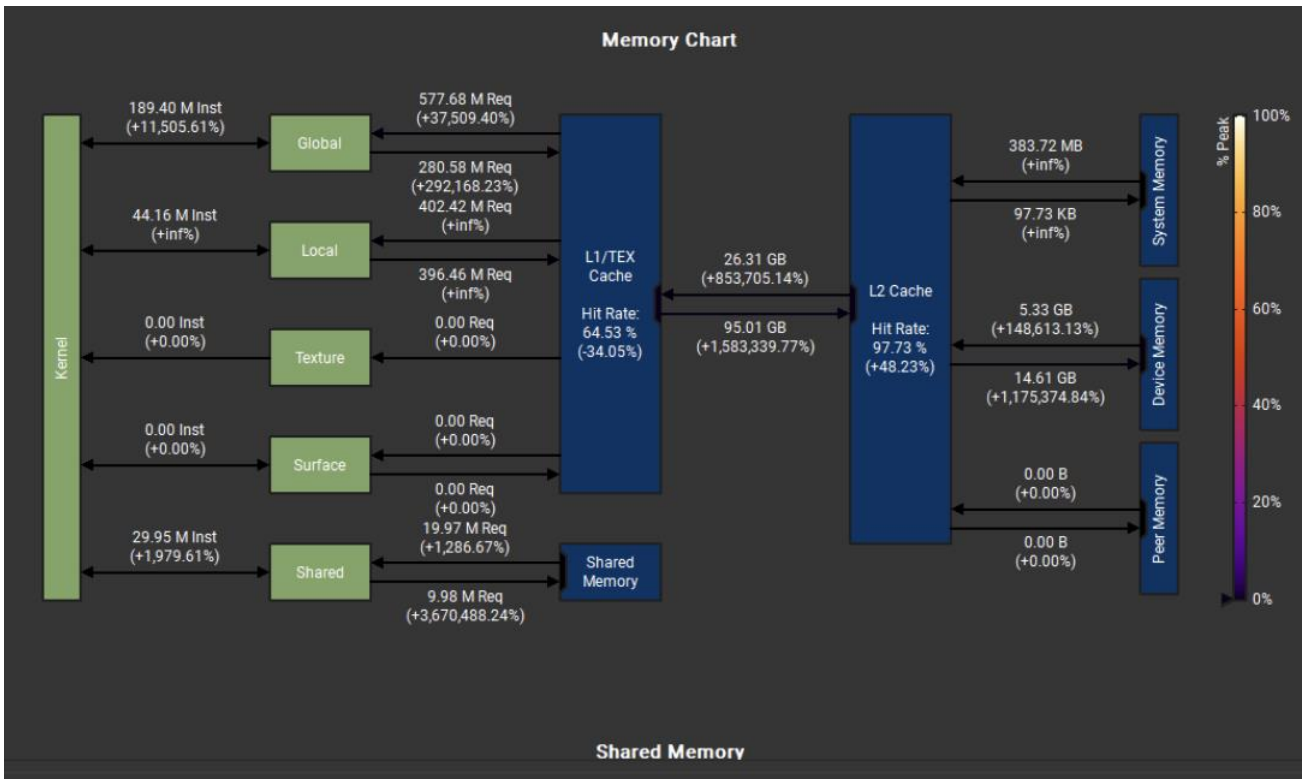

Fig. 30. Memory Chart for Parallelization by Dynamic

There are three instances where memory coalescing occurred. The first instance occurred when loading global memory to the tiles in the Mix Columns step. The second and third instance occurred when data from the plaintext is being loaded in the Add Round Key and Sub-bytes steps. The illustration for these three occurrances is shown in Fig. 31 – Fig. 33

```
for (int ph = 0; ph < ceil(width / (float)TILE_WIDTH); ++ph) {

    if ((row < width) && (ph * TILE_WIDTH + tx) < width)
    {
        ds_mix[ty][tx] = mixMatrix[row * width + ph * TILE_WIDTH + tx];
    }
    else
    {
        ds_mix[ty][tx] = 0;
    }
    if ((ph * TILE_WIDTH + ty) < width && col < width)
    {
        ds_state[ty][tx] = text[((ph * TILE_WIDTH + ty) * width + col) + index * 16];
    }
    else
    {
        ds_state[ty][tx] = 0;
    }

    __syncthreads();
```

Fig. 31. Memory Coalescing in Mix Columns

```
__global__ void addRoundKey(uint8_t* text, uint8_t* key, int index, int round) {

    int thread = threadIdx.x;

    if (thread < 16) {
        text[index * 16 + thread] ^= key[round * 16 + thread];
    }

}
```

Fig. 32. Memory Coalescing in Add Round Key

```
__global__ void subBytes(uint8_t* text, uint8_t* sBox, int index) {  nt round) {

    int thread = threadIdx.x;

    if (thread < 16) {
        text[index * 16 + thread] = sBox[text[index * 16 + thread]];
    }

}
```

Fig. 33. Memory Coalescing in Sub-bytes

As seen in Fig. 34, numerous requests have been made to the global memory due to the fact that virtually all data are stored in the global memory as compared to the Parallelization by State Array method where most data are stored in shared memory and registers. Although there are more memory coalescing for this method compared to the Parallelization by State Array method, the sectors/request value is still low, standing at 1.89. This may be due to the fact that many of the global memory accesses are still not coalesced. For instance, access to s-box values.

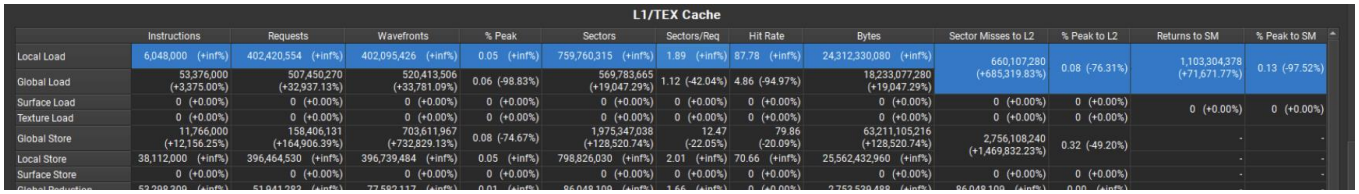

Fig. 34. L1/TEX Cache Information

As seen in Fig. 35, the kernel's theoretical occupancy is 50%. It is limited by the number of blocks that can fit on the SM. The achieved occupancy is 16.08 %, which is a big difference compared to the theoretical value. An occupancy of 16.08 % will result in latency during global loads.

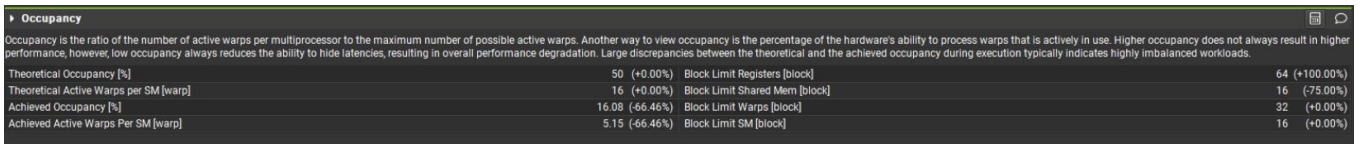

Fig. 35. Occupancy Information

VI. COMPARISON WITH SEQUENTIAL REFERENCE

Comparisons with the sequential reference will only be done with the Parallel by State Array method since it is more efficient, thus a better comparison to show the improvements made by applying parallelization.

The time complexity of the sequential algorithm is O(N), since Add Round Key, Sub-bytes, Shift Rows, and Mix Columns steps all only require O(N) time. N is the size of the plaintext. The time complexity of the Parallel by State Array algorithm is O(N/P), with P being the number of parallel "workers". According to the asymptotic speedup formula of

$S_p = \frac{T_1}{T_2}$, the **asymptotic speedup would be O(P**).

Results of successful runs:

- Key: Thats my Kung FuThats my Kung Fu
- Plaintext: Today I stayed home all day, it was a relaxing day.
- Output:

Sequential	Parallel By State Array
30 8 96 de 6f 8c 43 96 d4 97 81 b0 8 fd 7e ac 10 52 9d fc ba bd 2b 4b c3 d5 7b e8 d0 96 f 4 d2 aa 0 3f 70 e2 6b 2d 70 c2 75 cb 3e f1 ed 85 61 79 2e d 5 5 5 5 5 cd	30 8 96 de 6f 8c 43 96 d4 97 81 b0 8 fd 7e ac 10 52 9d fc ba bd 2b 4b c3 d5 7b e8 d0 96 f 4 d2 aa 0 3f 70 e2 6b 2d 70 c2 75 cb 3e f1 ed 85 61 79 2e d 5 5 5 5 5 cd

Table 1: Output values of Sequential and Parallel by State Array Methods

The Parallel by State Array algorithm is work efficient because the Add Round Key, Sub-bytes, Shift Rows, and Mix Columns steps are identical to the sequential algorithm. The main difference between them is that the

plaintext is split into many 16-byte state arrays for parallel execution in the Parallel by State Array method. Therefore, the parallel algorithm performs the same amount of work as the corresponding sequential algorithm.

VII. Conclusion

From the timing and Nsight Compute results, it is evident that the Parallel by State Array method is superior compared to the Parallel by Dynamic Parallelism method. This proves that storing frequently used data in shared memory and registers will result in faster execution compared with storing these data in global memory, although having individual steps in the encryption process parallelized.

Also, the results of this project show that the Parallel by State Array method performs much better than the sequential method when it comes to dataset larger than around 250kB. Thus, for small plaintext size, the sequential method is preferred. Whereas, for large plaintext size, the Parallel by State Array method is preferred. The reason that this occurs is because for small datasets, the time required for sequential execution is small compared to the overhead of having to allocate, send, and communicate data between host and device.