**고급소프트웨어실습 11주차 과제**

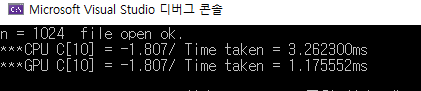
**:CUDA 프로그래밍의 기초1**

**20171672 이정원**

**실습 문제 3**

메인 메모리에서 두 배열 데이터를 읽어들여 y = Mxi 에 해당하는 백터 y를 구하는 프로그램이다.

* Init\_data() 함수와 init\_MatVec() 함수를 이용하여 입력 데이터를 설정한 후 Mat\_Vec\_Multiply()를 통해 CPU 계산을, Mat\_Vec\_Multiply\_GPU() 함수를 통해 GPU 계산을 처리한다. GPU 계산에서는 그리드를 1차원으로 설정하였고, CUDA API 함수를 이용해 device 메모리를 할당하였다. 특히 GPU 계산에 쓰이는 함수 Mat\_Vec\_Multiply\_Kernel() 함수는 병렬 계산을 위해 해당 thread의 아이디를 id 변수에 넣고, (blockDim.x \* blockIdx.x + threadIdx.x) 해당 thread의 결과 벡터를 구해 d\_VecY에 넣는다. 결과를 구한 후 다시 Host 메모리로 복사하여 결과 배열에 저장한다.
* 실행 결과



* 블록의 크기만큼 작업을 분할해 병렬 계산을 처리하는 GPU 계산의 속도가 빠른 것을 확인할 수 있다.

숙제 코드 파일(kernel.cu)은 각각 1번은 ‘숙제1번’ 폴더에, 2번은 ‘숙제2번’ 폴더에, 3번은 ‘숙제3번’ 폴더에 저장하였다.

**숙제 1**

2차 방정식의 근을 구해 주는 프로그램이다.

* **Init\_bin\_file**
* A.bin, B.bin, C.bin에 N\_EQUATIONS와 i번째 2차 방정식의 계수를 랜덤으로 생성하여 저장하는 함수이다.

|  |
| --- |
| void init\_bin\_file(int size) {  srand((unsigned)time(NULL));  printf("\*\*\*Binary File init Start!!\n");  FILE\* fp = fopen("A.bin", "wb");  fwrite(&size, sizeof(int), 1, fp);  int random\_num;  for (int i = 0; i < size; i++) {  random\_num = (float)((float)rand() / RAND\_MAX \* 20.0f - 10.0f);  fwrite(&random\_num, sizeof(float), 1, fp);  }  fclose(fp);  fp = fopen("B.bin", "wb");  fwrite(&size, sizeof(int), 1, fp);  for (int i = 0; i < size; i++) {  random\_num = (float)((float)rand() / RAND\_MAX \* 1000.0f + 100.0f);  fwrite(&random\_num, sizeof(float), 1, fp);  }  fclose(fp);  fp = fopen("C.bin", "wb");  fwrite(&size, sizeof(int), 1, fp);  for (int i = 0; i < size; i++) {  random\_num = (float)((float)rand() / RAND\_MAX \* 20.0f - 10.0f);  fwrite(&random\_num, sizeof(float), 1, fp);  }  fclose(fp);  printf("\*\*\*Binary File init End!!\n\n");  } |

* **Read\_bin\_file()**
* A.bin, B.bin, C.bin을 읽어 들여 배열에 저장하는 함수이다. 각각 A, B, C에 저장하였다.

|  |
| --- |
| void read\_bin\_file()  {  printf("\*\*\*Binary File Read Start!!\n");  FILE\* fp = fopen("A.bin", "rb");  /\*Todo\*/  fread(&N\_EQUATIONS, sizeof(int), 1, fp);  A = new float[N\_EQUATIONS];  B = new float[N\_EQUATIONS];  C = new float[N\_EQUATIONS];  X0 = new float[N\_EQUATIONS];  X1 = new float[N\_EQUATIONS];  FX0 = new float[N\_EQUATIONS];  FX1 = new float[N\_EQUATIONS];  X0\_GPU = new float[N\_EQUATIONS];  X1\_GPU = new float[N\_EQUATIONS];  FX0\_GPU = new float[N\_EQUATIONS];  FX1\_GPU = new float[N\_EQUATIONS];  fread(A, sizeof(float), N\_EQUATIONS, fp);  fclose(fp);  fp = fopen("B.bin", "rb");  fread(&N\_EQUATIONS, sizeof(int), 1, fp);  fread(B, sizeof(float), 1, fp);  fclose(fp);  fp = fopen("C.bin", "rb");  fread(&N\_EQUATIONS, sizeof(int), 1, fp);  fread(C, sizeof(float), 1, fp);  fclose(fp);  printf("\*\*\*Binary File Read End!!\n\n");  } |

* **Find\_roots\_CPU**
* 2차 방정식의 근을 찾아 X0, X1에 저장하는 CPU 코드이다. 함수의 결과는 FX0, FX1에 저장한다.

|  |
| --- |
| void find\_roots\_CPU(float\* A, float\* B, float\* C, float\* X0, float\* X1, float\* FX0, float\* FX1, int n) {  int i;  float a, b, c, d, x0, x1, tmp;  for (i = 0; i < n; i++) {  a = A[i]; b = B[i]; c = C[i];  d = sqrtf(b \* b - 4.0f \* a \* c);  tmp = 1.0f / (2.0f \* a);  X0[i] = x0 = (-b - d) \* tmp;  X1[i] = x1 = (-b + d) \* tmp;  FX0[i] = (a \* x0 + b) \* x0 + c;  FX1[i] = (a \* x1 + b) \* x1 + c;  }  } |

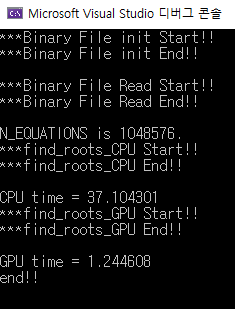
* **Find\_roots\_GPU**
* 2차 방정식의 근을 찾아 X0\_GPU, X1\_GPU에 저장하는 GPU 코드이다. 배열 복사를 위해 CUDA API 함수를 cudaMalloc, cudaMemcpy를 이용하여 device(GPU) 메모리에 메모리를 할당하고 host(CPU) 메모리에서 device 메모리로 메모리를 복사한다. 블록의 크기로 블록 안의 스레드 개수를 3차원적으로 지정해 주는 dimBlock은 (BLOCK\_SIZE, 1), 그리고 그리드의 크기로 그리드 안의 블록 개수를 3차원적으로 지정해 주는 dimGrid은 (N\_EQUATIONS / dimBlock.x, 1)로 설정해 준다. (2번째 인자 y가 1인 이유는 계산에서 쓰이는 배열이 1차원이기 때문, 3번째 인자 z는 생략됐으므로 자동으로 1로 지정이 됨) 이후 find\_roots\_Kernel 함수를 출력하여 커널을 수행시켜 각 thread에 대해 2차 방정식의 근과 함숫값을 계산해 준다.
* 시간 측정은 커널로 계산할 때만 필요하므로 해당 함수 앞뒤에 시간 측정 함수를 배치하였다.

|  |
| --- |
| cudaError\_t find\_roots\_GPU(float\* A, float\* B, float\* C, float\* X0\_GPU, float\* X1\_GPU, float\* FX0\_GPU, float\* FX1\_GPU)  {  CHECK\_TIME\_INIT\_GPU();  cudaError\_t cudaStatus;  cudaStatus = cudaSetDevice(0);  if (cudaStatus != cudaSuccess) {  fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  goto Error;  }///////////// if(cu..... ==CUDA\_CALL  /\*Todo\*/  float\* d\_A, \* d\_B, \* d\_C;  float\* d\_X0, \* d\_X1;  float\* d\_FX0, \* d\_FX1;  size\_t size;  size = N\_EQUATIONS \* sizeof(float);  CUDA\_CALL(cudaMalloc(&d\_A, size))  CUDA\_CALL(cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice))  CUDA\_CALL(cudaMalloc(&d\_B, size))  CUDA\_CALL(cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice))  CUDA\_CALL(cudaMalloc(&d\_C, size))  CUDA\_CALL(cudaMemcpy(d\_C, C, size, cudaMemcpyHostToDevice))  CUDA\_CALL(cudaMalloc(&d\_X0, size))  CUDA\_CALL(cudaMalloc(&d\_X1, size))  CUDA\_CALL(cudaMalloc(&d\_FX0, size))  CUDA\_CALL(cudaMalloc(&d\_FX1, size))    size = N\_EQUATIONS;  CHECK\_TIME\_START\_GPU()    dim3 dimBlock(BLOCK\_SIZE, 1);  dim3 dimGrid(N\_EQUATIONS / dimBlock.x, 1);  find\_roots\_Kernel << < dimGrid, dimBlock >> > (d\_A, d\_B, d\_C, d\_X0, d\_X1, d\_FX0, d\_FX1);  CHECK\_TIME\_END\_GPU(device\_time)  size = N\_EQUATIONS \* sizeof(float);  CUDA\_CALL(cudaGetLastError())  CUDA\_CALL(cudaDeviceSynchronize())  CUDA\_CALL(cudaMemcpy(X0\_GPU, d\_X0, size, cudaMemcpyDeviceToHost))  CUDA\_CALL(cudaMemcpy(X1\_GPU, d\_X1, size, cudaMemcpyDeviceToHost))  CUDA\_CALL(cudaMemcpy(FX0\_GPU, d\_FX0, size, cudaMemcpyDeviceToHost))  CUDA\_CALL(cudaMemcpy(FX1\_GPU, d\_FX1, size, cudaMemcpyDeviceToHost))    CHECK\_TIME\_DEST\_GPU();  Error:  cudaFree(d\_A);  cudaFree(d\_B);  cudaFree(d\_C);  cudaFree(d\_X0);  cudaFree(d\_X1);  cudaFree(d\_FX0);  cudaFree(d\_FX1);  return cudaStatus;  } |

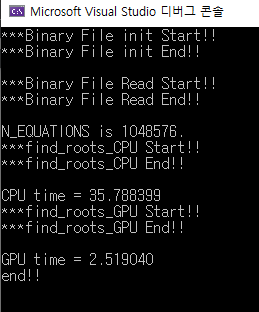
* **Find\_roots\_Kernel**
* 2차 방정식의 근을 계산해 d\_X0, d\_X1에 저장하는 커널 함수이다. 이때 tid는 blockDim.x\*blockIdx.x + threadIdx.x로 지정해 주었는데, 해당 thread에 대해 병렬적으로 계산해야 하므로 해당 thread의 인덱스를 tid에 지정한 것이다.

|  |
| --- |
| \_\_global\_\_ void find\_roots\_Kernel(float\* d\_A, float\* d\_B, float\* d\_C, float\* d\_X0, float\* d\_X1, float\* d\_FX0, float\* d\_FX1)  {  int tid;  tid = blockDim.x \* blockIdx.x + threadIdx.x;  float a, b, c, d, x0, x1, tmp;  a = d\_A[tid]; b = d\_B[tid]; c = d\_C[tid];  d = sqrtf(b \* b - 4.0f \* a \* c);  tmp = 1.0f / (2.0f \* a);  d\_X0[tid] = x0 = (-b - d) \* tmp;  d\_X1[tid] = x1 = (-b + d) \* tmp;  d\_FX0[tid] = (a \* x0 + b) \* x0 + c;  d\_FX1[tid] = (a \* x1 + b) \* x1 + c;  } |

* **실행 결과 (Debug 모드)**
* N\_EQUATIONS = 1048675 (실험하기 너무 크지도, 작지 않게 적합한 값을 임의로 설정하였다.)
* BLOCK\_SIZE = 32 / BLOCK\_SIZE = 64



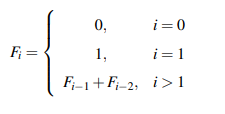
* BLOCK\_SIZE = 16 / BLOCK\_SIZE = 2



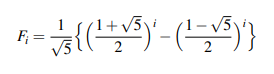
* CPU로 계산했을 때보다 GPU로 계산했을 때 모두 빠른 속도를 보였고, BLOCK\_SIZE가 작아질수록 GPU 계산 속도는 느려졌다. 블록의 크기는 작업을 분할하는 기준이므로 많은 수를 지정할수록 분할되는 작업이 많아지는 걸 의미하기 때문에 블록의 크기가 늘어날수록 실행 속도가 빨라진 것으로 보인다. 2, 16에서 32로 커졌을 때는 현저한 차이를 보였으나 32에서 64로 갔을 때는 근소하게 빨라진 것으로 보아 32, 64 등이 적절한 블록의 크기인 것으로 보인다.

**숙제 2**

숙제 2는 입력 배열에 저장되어 있는 N개의 임의의 정수 값 각각에 대한 피보나치 수를 계산하여 출력 배열에 저장하는 프로그램이다. 피보나치 수는 다음과 같이 정의된다.



임의의 n에 대해 Fn은 다음과 같이 표현할 수 있다.



* **Init\_bin\_file(int size)**
* 정수인 난수를 생성하여 x.bin 파일에 저장해 주는 함수이다.
* **Read\_bin\_file()**

|  |
| --- |
| void read\_bin\_file()  {  printf("\*\*\*Binary File Read Start!!\n");  FILE \*fp = fopen("x.bin", "rb");  /\*Todo\*/  fread(&N, sizeof(int), 1, fp);  h\_Fibonacci\_number = new int[N];  h\_fibonacci\_value = new int[N];  h\_fibonacci\_value\_GPU = new int[N];  fread(h\_Fibonacci\_number, sizeof(int), N, fp);  fclose(fp);  printf("\*\*\*Binary File Read End!!\n\n");  } |

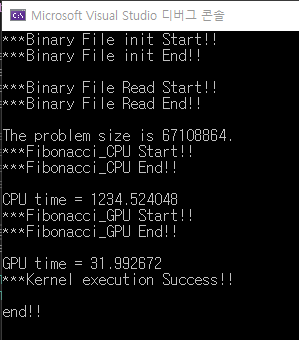
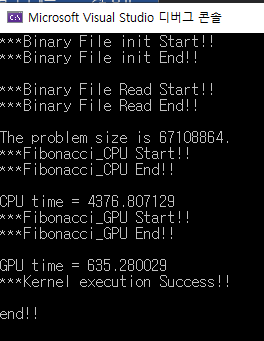
* x.bin에서 배열의 크기와 입력 배열을 읽어 들여 N, h\_Fibonacci\_number에 저장한다.
* **Fibonacci\_CPU(int \*h\_Fibo\_n, int \*h\_Fibo\_v)**
* 피보나치 수를 계산하는 CPU 코드이다. 결과는 h\_Fibo\_v에 저장한다.
* **Fibonacci\_GPU(int \*h\_Fibo\_n, int \*h\_Fibo\_v, int \*h\_Fibo\_v\_GPU)**

|  |
| --- |
| cudaError\_t Fibonacci\_GPU(int \*h\_Fibo\_n, int \*h\_Fibo\_v, int \*h\_Fibo\_v\_GPU)  {  CHECK\_TIME\_INIT\_GPU();  cudaError\_t cudaStatus;  cudaStatus = cudaSetDevice(0);  if (cudaStatus != cudaSuccess) {  fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  goto Error;  }///////////// if(cu..... ==CUDA\_CALL  /\*Todo\*/  int(\*d\_Fibo\_n), (\*d\_Fibo\_v);  size\_t size;    size = N \* sizeof(int);  CUDA\_CALL(cudaMalloc(&d\_Fibo\_n, size))  CUDA\_CALL(cudaMemcpy(d\_Fibo\_n, h\_Fibo\_n, size, cudaMemcpyHostToDevice))      CUDA\_CALL(cudaMalloc(&d\_Fibo\_v, size))  size = N;  CHECK\_TIME\_START\_GPU()  // Assume that width and height are multiples of BLOCK SIZE.  dim3 dimBlock(BLOCK\_SIZE, 1);  dim3 dimGrid(N / dimBlock.x, 1);  Fibonacci\_Kernel << < dimGrid, dimBlock >> > (d\_Fibo\_n, d\_Fibo\_v);    CHECK\_TIME\_END\_GPU(device\_time)  size = N \* sizeof(int);  CUDA\_CALL(cudaGetLastError())  // cudaDeviceSynchronize waits for the kernel to finish, and returns  // any errors encountered during the launch.  CUDA\_CALL(cudaDeviceSynchronize())  CUDA\_CALL(cudaMemcpy(h\_Fibo\_v\_GPU, d\_Fibo\_v, size, cudaMemcpyDeviceToHost))    CHECK\_TIME\_DEST\_GPU();  Error:  cudaFree(d\_Fibo\_n);  cudaFree(d\_Fibo\_v);    return cudaStatus;  } |

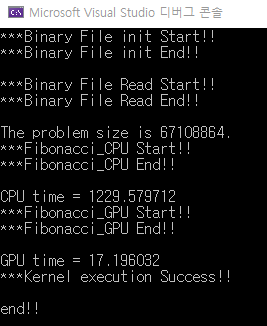
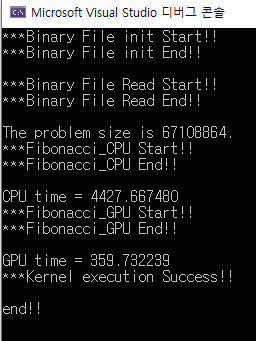
* 피보나치 수를 계산하는 GPU 코드이다. 입력 배열 복사를 위해 CUDA API 함수를 cudaMalloc, cudaMemcpy를 이용하여 device(GPU) 메모리에 메모리를 할당하고 host(CPU) 메모리에서 device 메모리로 메모리를 복사한다. (h\_Fibo\_n -> d\_Fibo\_n) 또한 출력 배열 d\_Fibo\_v도 cudaMalloc 함수를 이용하여 메모리 할당을 해 준다. 블록의 크기로 블록 안의 스레드 개수를 3차원적으로 지정해 주는 dimBlock은 (BLOCK\_SIZE, 1), 그리고 그리드의 크기로 그리드 안의 블록 개수를 3차원적으로 지정해 주는 dimGrid은 (N / dimBlock.x, 1)로 설정해 준다. (2번째 인자 y가 1인 이유는 계산에서 쓰이는 배열이 1차원이기 때문, 3번째 인자 z는 생략됐으므로 자동으로 1로 지정이 됨) 이후 Fibonacci\_Kernel 함수를 출력하여 커널을 수행시켜 각 thread에 대해 피보나치 수를 계산해 준다.
* 시간 측정은 커널로 계산할 때만 필요하므로 해당 함수 앞뒤에 시간 측정 함수를 배치하였다.
* **Fibonacci\_Kernel(int \*x, int \*y)**

|  |
| --- |
| \_\_global\_\_ void Fibonacci\_Kernel(int \*x, int\* y)  {  /\*Thread Idx\*/  int row;  int col;  int tid;  /\*Todo\*/  tid = blockDim.x \* blockIdx.x + threadIdx.x;  float sqrt\_5, x\_0, x\_1, tmp\_0, tmp\_1;  sqrt\_5 = sqrtf(5.0f);  x\_0 = (1.0f + sqrt\_5) / 2.0f; x\_1 = (1.0f - sqrt\_5) / 2.0f;  tmp\_0 = tmp\_1 = 1.0f;  for (int j = 0; j < x[tid]; j++) {  tmp\_0 \*= x\_0; tmp\_1 \*= x\_1;  }  y[tid] = (int)((tmp\_0 - tmp\_1) / sqrt\_5 + 0.5f);  } |

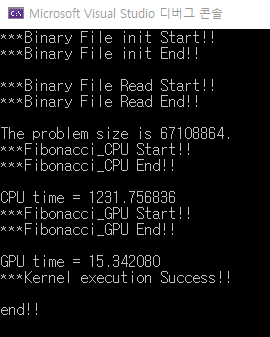
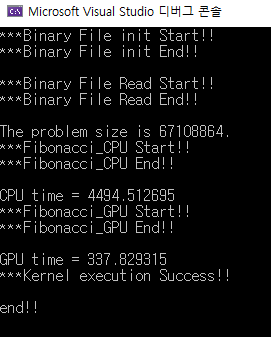
* 각 thread에 대해 피보나치 수를 계산하는 GPU 함수이다. tid를 설정한 후 해당 thread에 대한 출력 배열을 계산한다.
* **BLOCK\_SIZE에 대한 실행 결과 (왼쪽: Debug 모드, 오른쪽: Release 모드)**
* BLOCK\_SIZE 16



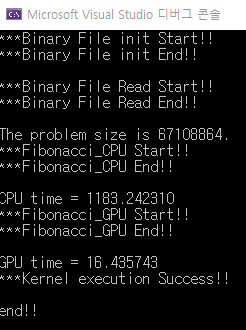
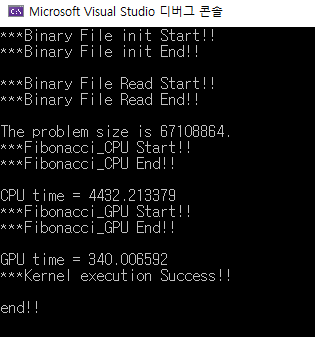
* BLOCK\_SIZE 32



* BLOCK\_SIZE 64



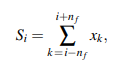
* BLOCK\_SIZE 128



* 모든 실행 결과를 통해 GPU 계산 속도가 CPU 계산 속도보다 현저히 빠르다는 것을 확인할 수 있었다. 동일한 방식의 계산이 서로 다른 데이터에 대하여 반복이 되고 있고, 그 계산이 서로 독립적일 때 CUDA 프로그램을 이용하여 병렬 계산을 수행할 수 있는데, 이 때문에 CPU보다 확실히 빠르게 계산을 완료할 수 있는 것이다. 또한, BLOCK\_SIZE가 커질수록 GPU 계산 속도가 점점 더 빨라지는 것을 확인할 수 있었다. 블록의 수는 작업을 분할하는 기준이므로 많은 수를 지정할수록 분할되는 작업이 많아지는 걸 의미하기 때문에 블록의 수가 늘어날수록 실행 속도가 빨라진 것으로 보인다. 그러나 32 이후로는 계산 속도가 비등비등한 결과로 나왔다.

**숙제 3**

-100과 100사이의 값을 가지는 n개의 정수로 구성된 수열 X에 대하여 다음과 같은 S 수열을 구하는 프로그램이다.



* **Read\_bin\_file(), init\_bin\_file()**
* 입력배열을 랜덤으로 생성해 Cuda\_HW3\_input.bin 파일에 저장하고 그것을 읽어들여와 h\_ArrayElements에 저장하는 함수들이다.
* **Sum\_n\_elements\_CPU**

|  |
| --- |
| void Sum\_n\_elements\_CPU(IN int \*p\_ArrayElements, OUT int \*p\_SumOfElements\_CPU, int Nf) {  for (int i = 0; i < N; i++) {  p\_SumOfElements\_CPU[i] = 0;  for (int k = i - Nf; k <= i + Nf; k++) {  if (k < 0 || k >= N) continue;  p\_SumOfElements\_CPU[i] += p\_ArrayElements[k];  }  }  } |

* 배열의 index-Nf부터 index+Nf 데이터까지의 합을 계산하는 CPU 프로그램이다. K가 0보다 작거나 N 이상일 경우 0으로 간주해 더하지 않고 다음으로 넘어간다. 출력 배열은 p\_SumOfElements\_CPU에 저장된다.
* **Sum\_n\_elements\_GPU**

|  |
| --- |
| cudaError\_t Sum\_n\_elements\_GPU(IN int \*p\_ArrayElements, OUT int \*p\_SumOfElements\_GPU, int Nf) {  cudaError\_t cudaStatus;  /\*Todo\*/  CHECK\_TIME\_INIT\_GPU();  cudaStatus = cudaSetDevice(0);  if (cudaStatus != cudaSuccess) {  fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");  goto Error;  }///////////// if(cu..... ==CUDA\_CALL  /\*Todo\*/  int(\*d\_Array), (\*d\_Sum);  size\_t size;  size = N \* sizeof(int);  CUDA\_CALL(cudaMalloc(&d\_Array, size))  CUDA\_CALL(cudaMemcpy(d\_Array, p\_ArrayElements, size, cudaMemcpyHostToDevice))  CUDA\_CALL(cudaMalloc(&d\_Sum, size))  size = N;  CHECK\_TIME\_START\_GPU()  // Assume that width and height are multiples of BLOCK SIZE.  dim3 dimBlock(BLOCK\_SIZE, 1);  dim3 dimGrid(N / dimBlock.x, 1);  Sum\_n\_elements\_Kernel << < dimGrid, dimBlock >> > (d\_Array, d\_Sum, N, Nf);  CHECK\_TIME\_END\_GPU(device\_time)    size = N \* sizeof(int);  CUDA\_CALL(cudaGetLastError())  // cudaDeviceSynchronize waits for the kernel to finish, and returns  // any errors encountered during the launch.  CUDA\_CALL(cudaDeviceSynchronize())  CUDA\_CALL(cudaMemcpy(p\_SumOfElements\_GPU, d\_Sum, size, cudaMemcpyDeviceToHost))  CHECK\_TIME\_DEST\_GPU();  Error:  cudaFree(d\_Array);  cudaFree(d\_Sum);  return cudaStatus;  } |

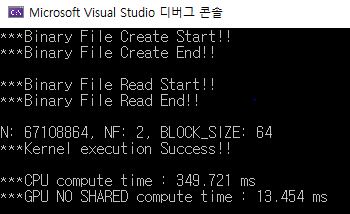
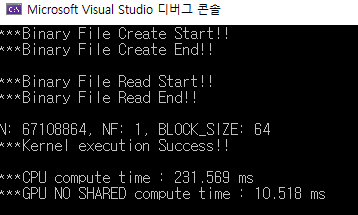
* 데이터를 계산하는 GPU 코드이다. 숙제 2와 같이 cudaMalloc, cudaMemcpy를 이용하여 d\_Array, d\_Sum을 device(GPU) 메모리에 할당한다. 이후 Sum\_n\_elements\_kernel 함수를 호출해 d\_Sum을 계산하여 넣는다. 숙제 2와 같이 1차원 그리드로 설정하였다.
* **Sum\_n\_elements\_Kernel**

|  |
| --- |
| \_\_global\_\_ void Sum\_n\_elements\_Kernel(IN int \*d\_ArrayElements, OUT int \*d\_SumOfArrayElements, int N, int Nf) {  int tid = blockDim.x \* blockIdx.x + threadIdx.x;  d\_SumOfArrayElements[tid] = 0;  for (int k = tid - Nf; k <= tid + Nf; k++) {  if (k < 0 || k >= N) continue;  d\_SumOfArrayElements[tid] += d\_ArrayElements[k];  }  } |

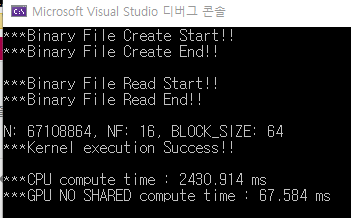
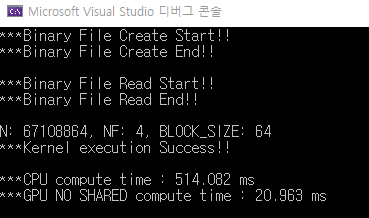
* D\_Sum을 계산하는 커널 함수이다.
* **실행 결과 (Release 모드)**

같은 BLOCK\_SIZE에 대하여 Nf가 다를 때

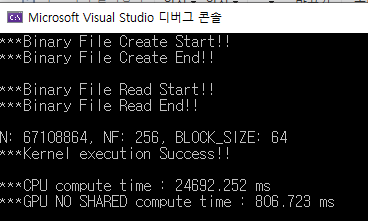
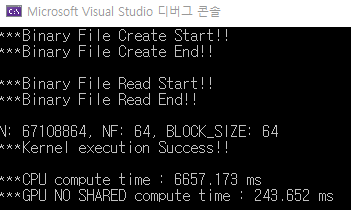
* Nf: 1, BLOCK\_SIZE: 64 / Nf: 2, BLOCK\_SIZE: 64



* Nf: 4, BLOCK\_SIZE: 64 / Nf: 16, BLOCK\_SIZE: 64



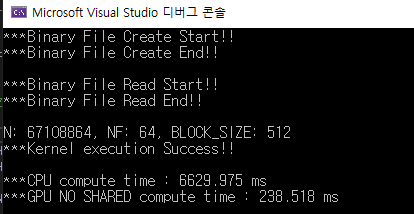
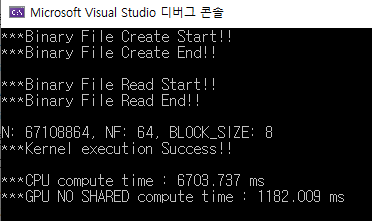
* Nf: 64, BLOCK\_SIZE: 64 / Nf: 256, BLOCK\_SIZE: 64



* Nf가 커질수록 CPU와 GPU 계산 속도는 현저히 증가하는 것을 확인할 수 있었다. Nf가 커질수록 합을 구하는 계산량도 늘어나기 때문에 처리하는 시간도 오래 걸리는 것으로 보인다.

같은 Nf에 대하여 BLOCK\_SIZE가 다를 때

* Nf: 64, BLOCK\_SIZE: 8 / Nf: 64, BLOCK\_SIZE: 512



* 숙제 2와 같이 블록의 수가 커질수록 GPU의 처리하는 시간도 줄어드는 것을 확인할 수 있었다. 블록의 수가 많아질수록 분배의 수가 많아지는 것이므로 처리 속도가 빨라진 것으로 보인다.