**고급소프트웨어실습1 12주차 과제**

컴퓨터공학 20172141 김미소

►Visual Studio 2019, CUDA 11.1 사용하였습니다.

**[실습1]**

**1. 먼저 for 문장을 사용하여 순차적으로 n번의 동일한 계산을 반복적으로 수행하는 C/C++ 함수 (CPU-Code)를 작성한 후 CPU 수행 시간을 측정하라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

**2. 이 문제를 해결해주는 CUDA 커널 프로그램 (GPU-Code-NO-SM)을 작성한 후 가급적 정확하게 GPU 수행 시간을 측정하라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

**3. 다음 위의 코드를 CUDA의 shared memory 기능을 사용하여 성능을 향상시켜주는 프로그램 (GPU-Code-SM)을 작성하라.**

텍스트이(가) 표시된 사진

자동 생성된 설명

**4. 두 프로그램 GPU-Code-NO-SM와 GPU-Code-SM의 성능 차이를 분석하기 위하여, 주어진 쓰레드 블럭 크기에 대하여 중복성을 의미하는 nf값을 변화시켜가면서, 두 버전의 시간을 측정한 후 왜 그러한 결과/차이가 산출되었는지 분석하라.**

N = 1<<26 즉, 226인 상태로

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BLOCK\_SIZE | nf | CPU(ms) | GPU NO SHARED(ms) | GPU SHARED(ms) | CPU/GPU NO SHARED | CPU/GPU SHARED |
| 16 | 1 | 165.883 | 12.208 | 12.394 | 13.58806 | 13.38414 |
| 8 | 684.886 | 14.373 | 18.815 | 47.65087 | 36.40106 |
| 16 | 1207.268 | 20.475 | 25.493 | 58.96303 | 47.35684 |
| 64 | 4871.375 | 56.431 | 81.825 | 86.32445 | 59.53407 |
| 128 | 9311.815 | 101.050 | 155.229 | 92.15057 | 59.9876 |
| 256 | 18198.572 | 165.564 | 303.815 | 109.9187 | 59.90018 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BLOCK\_SIZE | nf | CPU(ms) | GPU NO SHARED(ms) | GPU SHARED(ms) | CPU/GPU NO SHARED | CPU/GPU SHARED |
| 32 | 1 | 164.908 | 6.562 | 7.505 | 25.13075 | 21.97308 |
| 8 | 651.355 | 7.919 | 10.188 | 82.25218 | 63.93355 |
| 16 | 1207.942 | 10.924 | 15.184 | 110.5769 | 79.55361 |
| 64 | 4875.972 | 29.014 | 41.505 | 168.0558 | 117.4791 |
| 128 | 9345.148 | 52.676 | 84.611 | 177.4081 | 110.4484 |
| 256 | 18271.467 | 96.305 | 153.893 | 189.725 | 118.7284 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BLOCK\_SIZE | nf | CPU(ms) | GPU NO SHARED(ms) | GPU SHARED(ms) | CPU/GPU NO SHARED | CPU/GPU SHARED |
| 64 | 1 | 165.556 | 3.478 | 3.63 | 47.60092 | 45.60771 |
| 8 | 650.294 | 4.74 | 6.379 | 137.1928 | 101.9429 |
| 16 | 1205.501 | 6.863 | 10.172 | 175.6522 | 118.5117 |
| 64 | 4874.584 | 20.977 | 28.797 | 232.3776 | 169.274 |
| 128 | 9316.261 | 40.213 | 51.609 | 231.6729 | 180.5162 |
| 256 | 18196.986 | 81.529 | 101.446 | 223.1965 | 179.3761 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BLOCK\_SIZE | nf | CPU(ms) | GPU NO SHARED(ms) | GPU SHARED(ms) | CPU/GPU NO SHARED | CPU/GPU SHARED |
| 128 | 1 | 167.611 | 2.997 | 3.796 | 55.92626 | 44.15464 |
| 8 | 650.705 | 4.349 | 5.718 | 149.6218 | 113.7994 |
| 16 | 1205.194 | 6.238 | 8.179 | 193.202 | 147.3522 |
| 64 | 4877.695 | 20.265 | 26.101 | 240.6955 | 186.8777 |
| 128 | 9306.753 | 39.834 | 40.378 | 233.6384 | 230.4907 |
| 256 | 18189.412 | 79.223 | 75.333 | 229.5976 | 241.4534 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| BLOCK\_SIZE | nf | CPU(ms) | GPU NO SHARED(ms) | GPU SHARED(ms) | CPU/GPU NO SHARED | CPU/GPU SHARED |
| 256 | 1 | 164.551 | 3.012 | 3.652 | 54.63181 | 45.05778 |
| 8 | 650.925 | 4.508 | 5.175 | 144.3933 | 125.7826 |
| 16 | 1205.99 | 6.588 | 7.276 | 183.0586 | 165.749 |
| 64 | 4867.683 | 20.242 | 23.168 | 240.4744 | 210.1037 |
| 128 | 9313.517 | 38.893 | 33.792 | 239.4651 | 275.6131 |
| 256 | 18192.562 | 77.306 | 64.544 | 235.3318 | 281.8629 |

전체적으로 CPU 계산과 GPU 계산 간 시간 차이는 상당히 컸다. 그러나 대부분 shared memory를 쓰지 않는 GPU NO SHARED가 GPU SHARED보다 더 빨리 수행되는 것을 확인하였다. 값을 제대로 구한 반면에 속도가 더 늦게 나오는 것은 shared memory를 쓰는 kernel 함수에서 최적화가 덜 된 코드를 작성하여 생긴 문제일 수도 있다.

블록 크기가 커질수록 GPU NO SHARED와 GPU SHARED간 시간 차이가 줄어드는 것을 볼 수 있었고 GPU SHARED가 GPU NO SHARED보다 빨라지는 시점은 블록크기가 128일때부터이다. NF가 커질수록 메모리에 접근하는 횟수가 많아지기 때문에 global memory에 접근하는 것보다 shared memory에 접근하는 것이 시간적으로 유리하다.

**[숙제1]**

**1. CPU 코드**

|  |
| --- |
| void CFilteringView::GaussianFilteringCPU() {  m\_mfcImage1 = new CImage();  int r, c;  m\_result = -1;  m\_mfcImage1->Load("SaltPepper\_24bit.bmp");  LONG width1 = m\_mfcImage1->GetWidth();  LONG height1 = m\_mfcImage1->GetHeight();  WORD bitcount1 = m\_mfcImage1->GetBPP();  int pitch1 = m\_mfcImage1->GetPitch();  m\_width = width1;  m\_height = height1;  COLORREF col;  unsigned char gray;  m\_imageBits = new unsigned char[m\_width \* m\_height];  m\_imageBitsFiltered = new unsigned char[m\_width \* m\_height];  int max\_gray = 0;  int max\_d = 0;  // take R value only because R, G, B have the same value  for (r = 0; r < m\_height; r++) {  for (c = 0; c < m\_width; c++) {  col = m\_mfcImage1->GetPixel(c, r);  m\_imageBits[r \* m\_width + c] = GetRValue(col);  // GetGValue() and GetBValue() to take the multi-channel values  }  }  // Set gaussian kernel  float \_1 = 1.0f / 256.0f;  float \_4 = \_1 \* 4;  float \_6 = \_1 \* 6;  float \_16 = \_1 \* 16;  float \_24 = \_1 \* 24;  float \_36 = \_1 \* 36;  float\* p\_gaussian\_kernel = new float[25];  p\_gaussian\_kernel[0] = p\_gaussian\_kernel[4] = p\_gaussian\_kernel[20] = p\_gaussian\_kernel[24] = \_1;  p\_gaussian\_kernel[1] = p\_gaussian\_kernel[3] = p\_gaussian\_kernel[5] = p\_gaussian\_kernel[9] = \_4;  p\_gaussian\_kernel[15] = p\_gaussian\_kernel[19] = p\_gaussian\_kernel[21] = p\_gaussian\_kernel[23] = \_4;  p\_gaussian\_kernel[2] = p\_gaussian\_kernel[10] = p\_gaussian\_kernel[14] = p\_gaussian\_kernel[22] = \_6;  p\_gaussian\_kernel[6] = p\_gaussian\_kernel[8] = p\_gaussian\_kernel[16] = p\_gaussian\_kernel[18] = \_16;  p\_gaussian\_kernel[7] = p\_gaussian\_kernel[11] = p\_gaussian\_kernel[13] = p\_gaussian\_kernel[17] = \_24;  p\_gaussian\_kernel[12] = \_36;  QueryPerformanceFrequency(&m\_frequency);  QueryPerformanceCounter(&m\_start);  // Perform Gaussian filtering  double mean, temp;  int w0 = 5;  int w = floor((double)w0 / 2);  int k, l;    for (r = 0; r < m\_height; r++) {  for (c = 0; c < m\_width; c++) {  mean = 0;  for (int l = w \* -1; l <= w; l++) {  for (int k = w \* -1; k <= w; k++) {  if (r + k >= m\_height || r + k < 0 || c + l >= m\_width || c + l < 0) mean += 0;  else mean += m\_imageBits[(r + k) \* m\_width + c + l] \* p\_gaussian\_kernel[(w + k) \* w0 + w + l];  }  }  m\_imageBitsFiltered[r \* m\_width + c] = mean;  }  }  QueryPerformanceCounter(&m\_end);  m\_result = 1000 \* (m\_end.QuadPart - m\_start.QuadPart) / m\_frequency.QuadPart;  // Assign the same value into R, G, B channels.  for (r = 0; r < m\_height; r++) {  for (c = 0; c < m\_width; c++) {  //m\_mfcImage1->SetPixel(c,r,RGB(imageBits[r\*m\_width+c],imageBits[r\*m\_width+c],imageBits[r\*m\_width+c]));  temp = m\_imageBitsFiltered[r \* m\_width + c];  m\_mfcImage1->SetPixel(c, r, RGB(temp, temp, temp));  }  }  Invalidate();  delete m\_imageBits, m\_imageBitsFiltered;  delete[] p\_gaussian\_kernel;  } |

GaussianFilteringGPU 함수와 Set\_Gaussain\_Kernel 함수를 참고하여 작성하였다.

**2. Shared memory 비사용 GPU 코드**

|  |
| --- |
| \_\_global\_\_ void Gaussian\_kernel\_no\_shared(IN unsigned char \*d\_bitmaps, OUT unsigned char \*d\_Gaussian, long width, long height) {  const unsigned block\_id = blockIdx.y \* gridDim.x + blockIdx.x;  const unsigned thread\_id = threadIdx.y \* blockDim.x + threadIdx.x;  const unsigned id = block\_id \* BLOCK\_SIZE + thread\_id;  double mean = 0;  int w = 2;  int w0 = 5;  for (int c = 0; c < width; c++) {  mean = 0;  for (int l = -w; l <= w; l++) {  for (int k = -w; k <= w; k++) {  if (id + k >= height || id + k < 0 || c + l >= width || c + l < 0) mean += 0;  else mean += d\_bitmaps[(id + k) \* width + c + l] \* constant\_gaussian\_kernel[(w + k) \* w0 + w + l];  }  }  d\_Gaussian[id \* width + c] = mean;  }  } |

**3. Shared memory 사용 GPU 코드**

|  |
| --- |
| extern \_\_shared\_\_ unsigned char sharedBuffer[];  \_\_global\_\_ void Gaussian\_kernel\_shared(INOUT unsigned char \*d\_bitmaps, OUT unsigned char \*d\_Gaussian, long width, long height) {  const unsigned block\_id = blockIdx.y \* gridDim.x + blockIdx.x;  const unsigned thread\_id = threadIdx.y \* blockDim.x + threadIdx.x;  const unsigned id = block\_id \* BLOCK\_SIZE + thread\_id;    double mean = 0.0;  int w = floor((double)5 / 2);  int w0 = 5;  if (thread\_id == 0) {  for (int i = -w; i < 0; i++) {  for (int j = 0; j < width; j++) {  if (id + i < 0) sharedBuffer[(i + w) \* width + j] = 0;  else if (id + i <= height) sharedBuffer[(i + w) \* width + j] = d\_bitmaps[(id + i) \* width + j];  }  }  }  if (thread\_id == BLOCK\_SIZE - 1) {  for (int i = 1; i <= w; i++) {  for (int j = 0; j < width; j++) {  if (id + i >= height) sharedBuffer[(thread\_id + i + w) \* width + j] = 0;  else sharedBuffer[(thread\_id + i + w) \* width + j] = d\_bitmaps[(id + i) \* width + j];  }  }  }  for (int j = 0; j < width; j++) {  sharedBuffer[(thread\_id + w) \* width + j] = d\_bitmaps[id \* width + j];  }  \_\_syncthreads();  // syncrhonization of threas  for (int c = 0; c < width; c++) {  mean = 0.0;  for (int k = -w; k <= w; k++) {  for (int l = -w; l <= w; l++) {  if (c + l < 0 || c + l >= width) mean += 0;  else mean += sharedBuffer[(thread\_id + w + k) \* width + c + l] \* constant\_gaussian\_kernel[(w + k) \* w0 + w + l];  }  }  d\_Gaussian[(id)\*width + c] = mean;  }  } |

위 식들은



이 식을 참고하여 작성하였다.

- 측정 결과 (B1, B2)

텍스트, 회로, 전자기기이(가) 표시된 사진

자동 생성된 설명텍스트, 회로, 전자기기이(가) 표시된 사진

자동 생성된 설명BLOCK\_SIZE=32

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명BLOCK\_SIZE=2

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명BLOCK\_SIZE=8

텍스트, 전자기기이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명BLOCK\_SIZE=44

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 회로이(가) 표시된 사진

자동 생성된 설명BLOCK\_SIZE=88

- 측정 결과 (B3, B4)

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트, 장치이(가) 표시된 사진

자동 생성된 설명

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BLOCK\_SIZE | B1(ms) | B2(ms) | B3(ms) | B4(ms) |
| 32 | 14.69 | 13.62 | 18.00 | 9.00 |
| 2 | 14.04 | 13.62 | 18.00 | 9.00 |
| 8 | 14.59 | 13.60 | 18.00 | 9.00 |
| 44 | 14.61 | 13.15 | 18.00 | 9.00 |
| 88 | 14.59 | 14.39 | 18.00 | 9.00 |

기존 코드를 조금 수정해서 측정하였다. 블록 크기는 image height의 약수로 지정하였다.

블록 크기가 커질수록 수행시간이 빨라질 것으로 예상했으나 수행시간의 차이는 별로 없었다. 그리고 shared memory를 사용한 gpu계산이 shared memory를 사용하지 않은 gpu 계산보다 빠른 수행 시간을 보였다.