1. cusparse csc format

Step 1. Generate input matrices A,B

1) generate\_random\_dense\_matrix

Step 2. Create cusparse handle

1) cusparseCreate

Step 3. Allocate device memory

1) cudaMalloc

Step 4. Construct a descriptor of the matrix A

1) cusparseCreateMatDescr

Step 5. Set the matrix type field and indexbase field of the matrix descriptor

1) cusparseSetMatType

2) cusparseSetMatIndexBase

Step 6. Transfer the input matrices to the device and memset the output matrix

1) cudaMemcpy

2) cudaMemse

Step 7. Compute the number of non-zero elements in A and compute the number of non-zero elements in A per column

1) cusparseDnnz

Step 8. Allocate device memory to store the sparse CSC representation of A

1) cudaMalloc

Step 9. convert A from a dense formatting to a csc formatting.

1) cusparseDdense2csc

Step10. There is no csc function that can directly computes matrix-matrix mulplication with csc formatting. But we can achieve multiplication using the fact that csc is a csr transposed.

2) cusparseDcsrmm(handle, CUSPARSE\_OPERATION\_TRANSPOSE, N,

M, M, totalANnz, &alpha, Adescr, dCscValA,

dCscColPtrA,dCscRowIndA, dB, N, &beta, dC,

M)

So call this fuction with paremeter like this. Using CUSPARSE\_OPERATION\_TRANSPOSE, we can pass the transposed sparse matrix. And match the remain parameter to the transposed one.

Step 11. Get a result vector back to the host

1) cudaMemcpy

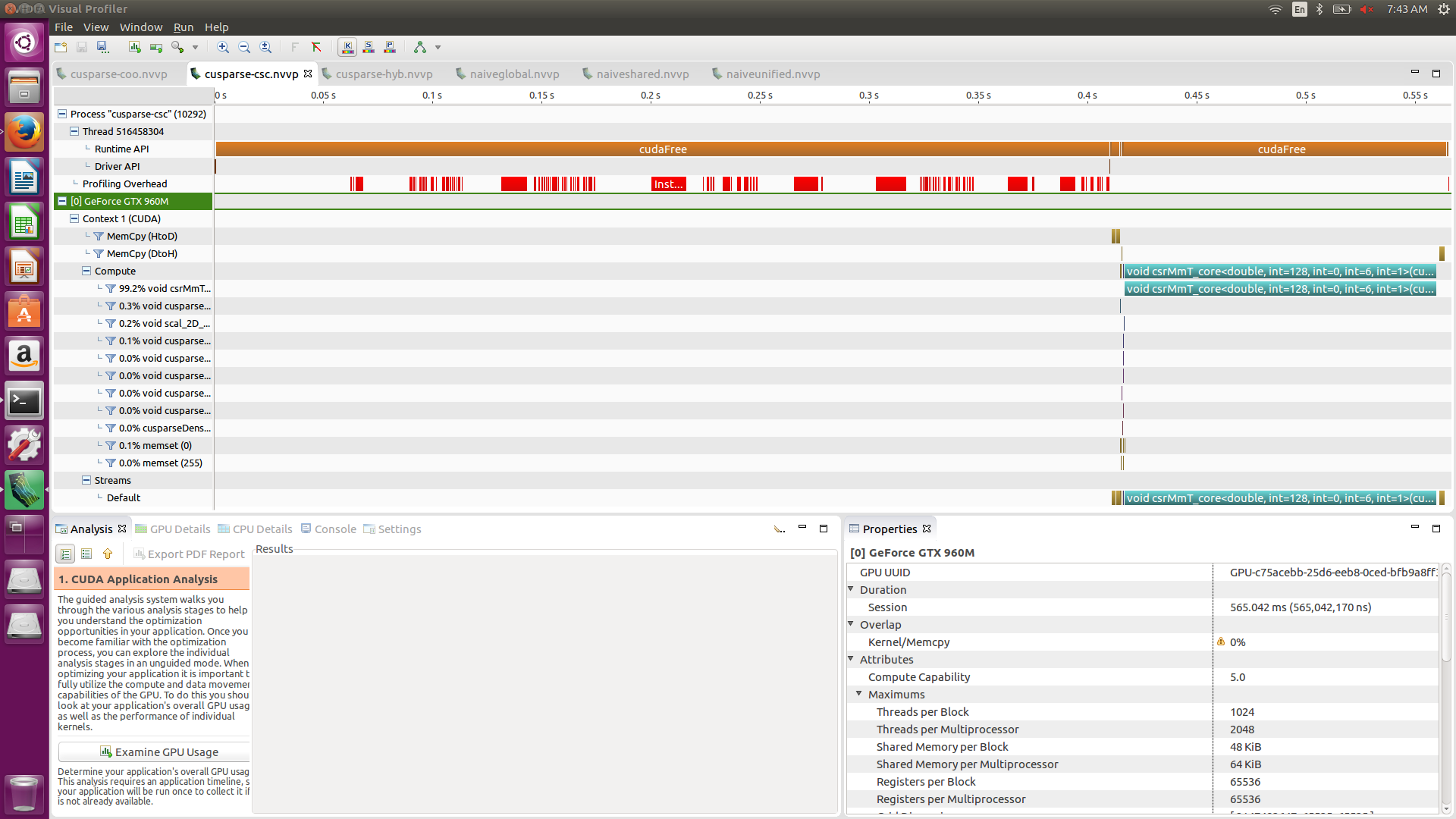
Step 12. Free the memory and matrix descriptor and resources

1) cudaFree

2) cusparseDestroyMatDescr

3) cusparseDestroy

[screen shot]



[source code]

#include "common.h"

#include <stdio.h>

#include <stdlib.h>

#include <cusparse\_v2.h>

#include <cuda.h>

/\*

\* M = # of rows

\* N = # of columns

\*/

int M = 1024;

int N = 1024;

/\*

\* Generate random dense matrix A in column-major order, while rounding some

\* elements down to zero to ensure it is sparse.

\*/

int generate\_random\_dense\_matrix(int M, int N, double \*\*outA)

{

int i, j;

double rMax = (double)RAND\_MAX;

double \*A = (double \*)malloc(sizeof(double) \* M \* N);

int totalNnz = 0;

for (j = 0; j < N; j++)

{

for (i = 0; i < M; i++)

{

int r = rand();

double \*curr = A + (j \* M + i);

if (r % 3 > 0)

{

\*curr = 0.0f;

}

else

{

double dr = (double)r;

\*curr = (dr / rMax) \* 100.0;

}

if (\*curr != 0.0f)

{

totalNnz++;

}

}

}

\*outA = A;

printf("totalNnz : %d\n",totalNnz);

return totalNnz;

}

// printf the partial matrix

// return : void

// parameter : double \* M - input matrix , int nrows - number of row in matrix, int ncols - number of column in matrix, int max\_row - the max row to be printed, int max\_col - the max col to be printed

void print\_partial\_matrix(double \*M, int nrows, int ncols, int max\_row,

int max\_col)

{

int row, col;

for (row = 0; row < max\_row; row++)

{

for (col = 0; col < max\_col; col++)

{

printf("%2.2lf ", M[row \* ncols + col]);

}

printf("...\n");

}

printf("...\n");

}

int main(int argc, char \*\*argv)

{

double \*A, \*dA;

double \*B, \*dB;

double \*C, \*dC;

int \*dANnzPerColumn;

double \*dCscValA;

int \*dCscRowIndA;

int \*dCscColPtrA;

int totalANnz;

double alpha = 1.0f;

double beta = 4.0f;

cusparseHandle\_t handle = 0;

cusparseMatDescr\_t Adescr = 0;

// Generate input

srand(9384);

int trueANnz = generate\_random\_dense\_matrix(M, N, &A);

int trueBNnz = generate\_random\_dense\_matrix(N, M, &B);

C = (double \*)malloc(sizeof(double) \* M \* M);

// print partial matrix

printf("A:\n");

print\_partial\_matrix(A, M, N, 10, 10);

printf("B:\n");

print\_partial\_matrix(B, N, M, 10, 10);

// Create the cuSPARSE handle

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t \* handle

// initialize the cuSPARSE library and creates a handle on the cuSPARSE context. it allocates hardware resources necessary for accessing the GPU

CHECK\_CUSPARSE(cusparseCreate(&handle));

// Allocate device memory for vectors and the dense form of the matrix A

// return : cudaError\_t

// parameter : void\*\* devPtr, size\_t size

// allocates (size) bytes of lineare memory on the device and returns in (\*devPtr) a pointer to the allocated memory

CHECK(cudaMalloc((void \*\*)&dA, sizeof(double) \* M \* N));

CHECK(cudaMalloc((void \*\*)&dB, sizeof(double) \* N \* M));

CHECK(cudaMalloc((void \*\*)&dC, sizeof(double) \* M \* M));

CHECK(cudaMalloc((void \*\*)&dANnzPerColumn, sizeof(int) \* M));

// Construct a descriptor of the matrix A

// return : cusparseStatus\_t

// parameter : cudsparseMatDescr\_t \* descrA

// the function initialize the matrix descriptor.

// It sets the fields MatrixType and IndexBase to the default values CUSPARSE\_MATRIX\_TYPE\_GENERAL and CUSPARSE\_INDEX\_BASE\_ZERO

CHECK\_CUSPARSE(cusparseCreateMatDescr(&Adescr));

// set the MatrixType field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA, cusparseMatrixType\_t type

CHECK\_CUSPARSE(cusparseSetMatType(Adescr, CUSPARSE\_MATRIX\_TYPE\_GENERAL));

// set the IndexBase field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA , cusparseIndexBase\_t base

CHECK\_CUSPARSE(cusparseSetMatIndexBase(Adescr, CUSPARSE\_INDEX\_BASE\_ZERO));

// Transfer the input vectors and dense matrix A to the device

// return : cudaError\_t

// parameter : void\* dst, const void\* src, size\_t count, enum cudaMemcpyKind kind

// copies (count) bytes from the memory area pointed by (src) to the memory area pointed to by (dst), where (kind) is one of enum type

CHECK(cudaMemcpy(dA, A, sizeof(double) \* M \* N, cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(dB, B, sizeof(double) \* N \* M, cudaMemcpyHostToDevice));

CHECK(cudaMemset(dC, 0x00, sizeof(double) \* M \* M));

// Compute the number of non-zero elements in A and compute the number of non-zero elements in A per column

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle , cusparseDirection\_t dirA, int m, int n, const cusparseMatDescr\_t descrA, const double \*A, int lda, int\* nnzPerRowColumn, int\* nnzTotalDevHostPtr

// input : handle - handle to the cuSPARSE library context

// dirA - direction that specifies whether to count nonzero elements by CUSPARSE\_DIRECTION\_ROW or by CUSPARSE\_DIRECTION\_COLUMN

// m - number of rows of matrix A

// n - number of columns of matrix A

// descrA - the descriptor of matrix A

// A - array of dimensions (lda,n)

// lda - leading dimension of dense array A

// output: nnzPerRowColumn - array of size m

// : nnzTotalDevHostPtr - total number of nonzero elements in device or host memory

CHECK\_CUSPARSE(cusparseDnnz(handle, CUSPARSE\_DIRECTION\_COLUMN, M, N, Adescr,

dA, M, dANnzPerColumn, &totalANnz));

if (totalANnz != trueANnz)

{

fprintf(stderr, "Difference detected between cuSPARSE NNZ and true "

"value: expected %d but got %d\n", trueANnz, totalANnz);

return 1;

}

// Allocate device memory to store the sparse CSC representation of A

// return : cudaError\_t

// parameter : void\*\* devPtr, size\_t size

// allocates (size) bytes of lineare memory on the device and returns in (\*devPtr) a pointer to the allocated memory

CHECK(cudaMalloc((void \*\*)&dCscValA, sizeof(double) \* totalANnz));

CHECK(cudaMalloc((void \*\*)&dCscRowIndA, sizeof(int) \* totalANnz));

CHECK(cudaMalloc((void \*\*)&dCscColPtrA, sizeof(int) \* (N + 1)));

// Convert A from a dense formatting to a CSC formatting, using the GPU

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle, int m, int n, const cusparseMatDescr\_t descA, const double \*A, int lda, const int \*nnzPerCol, double \*cscValA, int\* cscRowIndA,int\* cscColPtrA

CHECK\_CUSPARSE(cusparseDdense2csc(handle, M, N, Adescr, dA, M, dANnzPerColumn,

dCscValA, dCscRowIndA, dCscColPtrA));

// Perform matrix-matrix multiplication with the CSC-formatted matrix A

// csc is csr transposed. so we can use csr matrix multiplication with CUSPARSE\_OPERATION\_TRANSPOSE

// parameter : cusparseHandle\_t handle, cusparseOperation\_t transA, int m, int n, int k, int nnz, const double \*alpha,

// const cusparseMatDescr\_t descrA, const double \*csrValA, const int \*csrRowPtrA, const int \*csrColIndA,

// const double \*B, int ldb, const double \*beta, double \*C, int ldc

CHECK\_CUSPARSE(cusparseDcsrmm(handle, CUSPARSE\_OPERATION\_TRANSPOSE, N,

M, M, totalANnz, &alpha, Adescr, dCscValA,

dCscColPtrA,dCscRowIndA, dB, N, &beta, dC,

M));

// Copy the result vector back to the host

CHECK(cudaMemcpy(C, dC, sizeof(double) \* M \* M, cudaMemcpyDeviceToHost));

// print the result partial matrix

printf("C:\n");

print\_partial\_matrix(C, M, M, 10, 10);

// free the memory

free(A);

free(B);

free(C);

CHECK(cudaFree(dA));

CHECK(cudaFree(dB));

CHECK(cudaFree(dC));

CHECK(cudaFree(dANnzPerColumn));

CHECK(cudaFree(dCscValA));

CHECK(cudaFree(dCscRowIndA));

CHECK(cudaFree(dCscColPtrA));

// releases the memory allocated for the matrix descriptor

// return : cusparseStatus\_t

// parameter : cusparseDescr\_t descrA

CHECK\_CUSPARSE(cusparseDestroyMatDescr(Adescr));

// releases CPU-side resources used by the cuSPARSE library.

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle

CHECK\_CUSPARSE(cusparseDestroy(handle));

return 0;

}

2. cusparse coo format

Step 1. Generate input matrices A,B

1) generate\_random\_dense\_matrix

Step 2. Create cusparse handle

1) cusparseCreate

Step 3. Allocate device memory

1) cudaMalloc

Step 4. Construct a descriptor of the matrix A

1) cusparseCreateMatDescr

Step 5. Set the matrix type field and indexbase field of the matrix descriptor

1) cusparseSetMatType

2) cusparseSetMatIndexBase

Step 6. Transfer the input matrices to the device and memset the output matrix

1) cudaMemcpy

2) cudaMemse

Step 7. Compute the number of non-zero elements in A and compute the number of non-zero elements in A per row

1) cusparseDnnz

Step 8. Allocate device memory to store the sparse CSC & COO representation of A

1) cudaMalloc

Step 9. there is no coo function that can convert A from a dense format into a coo format. to achieve this, first convert to csr format, and then convert to coo format

1) cusparseDdense2csr : convert A from a dense format into a csr format

2) cusparseXcsr2coo : convert csr to coo format

Step 10. there is no coo matrix-matrix multiplication and matrix-vector multiplication.

So, I just use matrix-matrix multiplication with csr format cause we already have it.

1) cusparseDcsrmm : matrix-matrix multiplication with csr format

Step 11. Get a result vector back to the host

1) cudaMemcpy

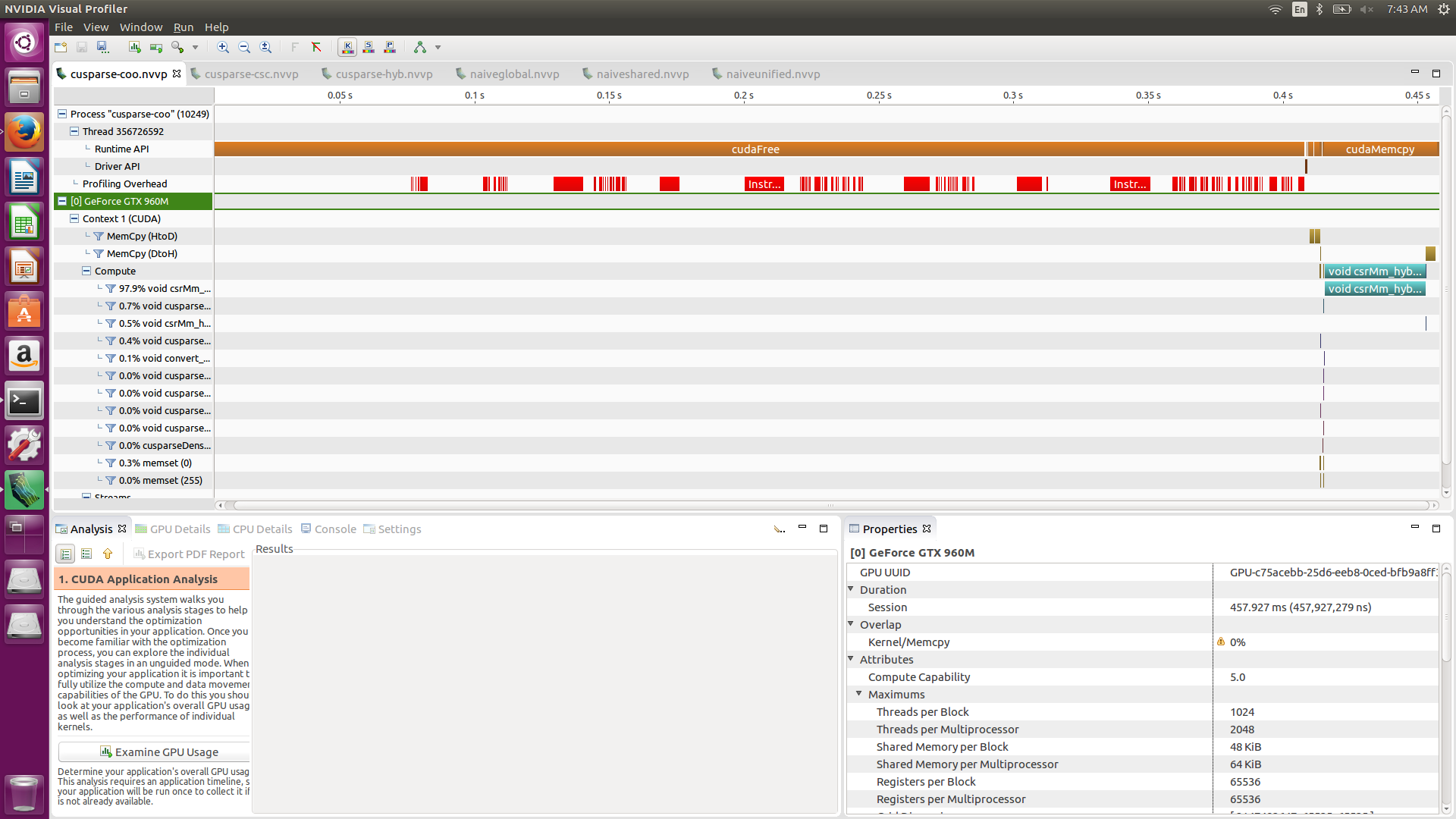
Step 12. Free the memory and matrix descriptor and resources

1) cudaFree

2) cusparseDestroyMatDescr

3) cusparseDestroy

[screen shot]



[source code]

#include "common.h"

#include <stdio.h>

#include <stdlib.h>

#include <cusparse\_v2.h>

#include <cuda.h>

/\*

\* This is an example demonstrating usage of the cuSPARSE library to perform a

\* sparse matrix-vector multiplication on randomly generated data.

\*/

/\*

\* M = # of rows

\* N = # of columns

\*/

int M = 1024;

int N = 1024;

/\*

\* Generate random dense matrix A in column-major order, while rounding some

\* elements down to zero to ensure it is sparse.

\*/

int generate\_random\_dense\_matrix(int M, int N, double \*\*outA)

{

int i, j;

double rMax = (double)RAND\_MAX;

double \*A = (double \*)malloc(sizeof(double) \* M \* N);

int totalNnz = 0;

for (j = 0; j < N; j++)

{

for (i = 0; i < M; i++)

{

int r = rand();

double \*curr = A + (j \* M + i);

if (r % 3 > 0)

{

\*curr = 0.0;

}

else

{

double dr = (double)r;

\*curr = (dr / rMax) \* 100.0;

}

if (\*curr != 0.0)

{

totalNnz++;

}

}

}

\*outA = A;

return totalNnz;

}

// printf the partial matrix

// return : void

// parameter : double \* M - input matrix , int nrows - number of row in matrix, int ncols - number of column in matrix, int max\_row - the max row to be printed, int max\_col - the max col to be printed

void print\_partial\_matrix(double \*M, int nrows, int ncols, int max\_row,

int max\_col)

{

int row, col;

for (row = 0; row < max\_row; row++)

{

for (col = 0; col < max\_col; col++)

{

printf("%2.2f ", M[row \* ncols + col]);

}

printf("...\n");

}

printf("...\n");

}

int main(int argc, char \*\*argv)

{

double \*A, \*dA;

double \*B, \*dB;

double \*C, \*dC;

int \*dANnzPerRow;

double \*dCooValA;

int \*dCsrRowPtrA;

int \*dCooRowIndA;

int \*dCooColIndA;

int totalANnz;

double alpha = 1.0;

double beta = 4.0;

cusparseHandle\_t handle = 0;

cusparseMatDescr\_t Adescr = 0;

// Generate input

srand(9384);

int trueANnz = generate\_random\_dense\_matrix(M, N, &A);

int trueBNnz = generate\_random\_dense\_matrix(N, M, &B);

C = (double \*)malloc(sizeof(double) \* M \* M);

// print partial matrix

printf("A:\n");

print\_partial\_matrix(A, M, N, 10, 10);

printf("B:\n");

print\_partial\_matrix(B, N, M, 10, 10);

// Create the cuSPARSE handle

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t \* handle

// initialize the cuSPARSE library and creates a handle on the cuSPARSE context. it allocates hardware resources necessary for accessing the GPU

CHECK\_CUSPARSE(cusparseCreate(&handle));

// Allocate device memory for vectors and the dense form of the matrix A

// return : cudaError\_t

// parameter : void\*\* devPtr, size\_t size

// allocates (size) bytes of lineare memory on the device and returns in (\*devPtr) a pointer to the allocated memory

CHECK(cudaMalloc((void \*\*)&dA, sizeof(double) \* M \* N));

CHECK(cudaMalloc((void \*\*)&dB, sizeof(double) \* N \* M));

CHECK(cudaMalloc((void \*\*)&dC, sizeof(double) \* M \* M));

CHECK(cudaMalloc((void \*\*)&dANnzPerRow, sizeof(int) \* M));

// Construct a descriptor of the matrix A

// return : cusparseStatus\_t

// parameter : cudsparseMatDescr\_t \* descrA

// the function initialize the matrix descriptor.

// It sets the fields MatrixType and IndexBase to the default values CUSPARSE\_MATRIX\_TYPE\_GENERAL and CUSPARSE\_INDEX\_BASE\_ZERO

CHECK\_CUSPARSE(cusparseCreateMatDescr(&Adescr));

// set the MatrixType field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA, cusparseMatrixType\_t type

CHECK\_CUSPARSE(cusparseSetMatType(Adescr, CUSPARSE\_MATRIX\_TYPE\_GENERAL));

// set the IndexBase field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA , cusparseIndexBase\_t base

CHECK\_CUSPARSE(cusparseSetMatIndexBase(Adescr, CUSPARSE\_INDEX\_BASE\_ZERO));

// Transfer the input vectors and dense matrix A to the device

// return : cudaError\_t

// parameter : void\* dst, const void\* src, size\_t count, enum cudaMemcpyKind kind

// copies (count) bytes from the memory area pointed by (src) to the memory area pointed to by (dst), where (kind) is one of enum type

CHECK(cudaMemcpy(dA, A, sizeof(double) \* M \* N, cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(dB, B, sizeof(double) \* N \* M, cudaMemcpyHostToDevice));

CHECK(cudaMemset((void\*)dC, 0, sizeof(double) \* M \* M));

// Compute the number of non-zero elements in A and compute the number of non-zero elements in A per row

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle , cusparseDirection\_t dirA, int m, int n, const cusparseMatDescr\_t descrA, const double \*A, int lda, int\* nnzPerRowColumn, int\* nnzTotalDevHostPtr

// input : handle - handle to the cuSPARSE library context

// dirA - direction that specifies whether to count nonzero elements by CUSPARSE\_DIRECTION\_ROW or by CUSPARSE\_DIRECTION\_COLUMN

// m - number of rows of matrix A

// n - number of columns of matrix A

// descrA - the descriptor of matrix A

// A - array of dimensions (lda,n)

// lda - leading dimension of dense array A

// output: nnzPerRowColumn - array of size m

// : nnzTotalDevHostPtr - total number of nonzero elements in device or host memory

CHECK\_CUSPARSE(cusparseDnnz(handle, CUSPARSE\_DIRECTION\_ROW, M, N, Adescr,

dA, M, dANnzPerRow, &totalANnz));

if (totalANnz != trueANnz)

{

fprintf(stderr, "Difference detected between cuSPARSE NNZ and true "

"value: expected %d but got %d\n", trueANnz, totalANnz);

return 1;

}

// Allocate device memory to store the sparse CSR representation of A

// return : cudaError\_t

// parameter : void\*\* devPtr, size\_t size

// allocates (size) bytes of lineare memory on the device and returns in (\*devPtr) a pointer to the allocated memory

CHECK(cudaMalloc((void \*\*)&dCooValA, sizeof(double) \* totalANnz));

CHECK(cudaMalloc((void \*\*)&dCsrRowPtrA, sizeof(int) \* (M + 1)));

CHECK(cudaMalloc((void \*\*)&dCooColIndA, sizeof(int) \* totalANnz));

CHECK(cudaMalloc((void \*\*)&dCooRowIndA, sizeof(int) \* totalANnz));

// there is no coo function that can convert A from a dense formatting into a coo formatiing. so first convert to csr format

// Convert A from a dense formatting to a CSR formatting, using the GPU

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle, int m, int n, const cusparseMatDescr\_t descA, const double \*A, int lda, const int \*nnzPerCol, double \*cscValA, int\* cscRowIndA,int\* cscColPtrA

CHECK\_CUSPARSE(cusparseDdense2csr(handle, M, N, Adescr, dA, M, dANnzPerRow,

dCooValA, dCsrRowPtrA, dCooColIndA));

// conver csr format to coo format

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle, const int \* csrRowPtr, int nnz, int m , int \* cooRowInd, cusparseIndexBase\_t idxBase

CHECK\_CUSPARSE(cusparseXcsr2coo(handle,dCsrRowPtrA,totalANnz,M,dCooRowIndA,CUSPARSE\_INDEX\_BASE\_ZERO));

// Perform matrix-matrix multiplication with the CSR-formatted matrix A cause there is no coo matrix-matrix multiplication and matrix-vector multiplication

// parameter : cusparseHandle\_t handle, cusparseOperation\_t transA, int m, int n, int k, int nnz, const double \*alpha,

// const cusparseMatDescr\_t descrA, const double \*csrValA, const int \*csrRowPtrA, const int \*csrColIndA,

// const double \*B, int ldb, const double \*beta, double \*C, int ldc

CHECK\_CUSPARSE(cusparseDcsrmm(handle, CUSPARSE\_OPERATION\_NON\_TRANSPOSE, M,

M, N, totalANnz, &alpha, Adescr, dCooValA,

dCsrRowPtrA, dCooColIndA, dB, N, &beta, dC,

M));

// Copy the result vector back to the host

CHECK(cudaMemcpy(C, dC, sizeof(double) \* M \* M, cudaMemcpyDeviceToHost));

// print the result partial matrix

printf("C:\n");

print\_partial\_matrix(C, M, M,10,10);

// free the memory

free(A);

free(B);

free(C);

CHECK(cudaFree(dA));

CHECK(cudaFree(dB));

CHECK(cudaFree(dC));

CHECK(cudaFree(dANnzPerRow));

CHECK(cudaFree(dCooValA));

CHECK(cudaFree(dCsrRowPtrA));

CHECK(cudaFree(dCooColIndA));

CHECK(cudaFree(dCooRowIndA));

// releases the memory allocated for the matrix descriptor

// return : cusparseStatus\_t

// parameter : cusparseDescr\_t descrA

CHECK\_CUSPARSE(cusparseDestroyMatDescr(Adescr));

// releases CPU-side resources used by the cuSPARSE library.

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle

CHECK\_CUSPARSE(cusparseDestroy(handle));

return 0;

}

3. cusparse hyb formatting

Step 1. Generate input matrices A,B

1) generate\_random\_dense\_matrix

Step 2. Create cusparse handle

1) cusparseCreate

Step 3. Allocate device memory

1) cudaMalloc

Step 4. Construct a descriptor of the matrix A

1) cusparseCreateMatDescr

Step 5. Set the matrix type field and indexbase field of the matrix descriptor

1) cusparseSetMatType

2) cusparseSetMatIndexBase

Step 6. Create and initialize the hybrid opaque data structure

1) cusparseCreateHybMa

Step 7. Transfer the input matrices to the device and memset the output matrix

1) cudaMemcpy

2) cudaMemse

Step 8. Compute the number of non-zero elements in A and compute the number of non-zero elements in A per row

1) cusparseDnnz

Step 9. convert A from a dense formatting to a hyb formatting.

1) cusparseDdense2hyb

Step10. There is no hyb function that can directly computes matrix-matrix mulplication. But there is a function that performs the matrix-vector multiplication.

1) cusparseDhybmv

So call this function several times, we can get a matrix-matrix multiplication.

In cusparse. Matrix-matrix multiplication can be done like this.

Sparse matrix (index mean linear address in array)

|  |  |
| --- | --- |
| Idx 0 | Idx 1 |
| Idx 2 | Idx 3 |

Dense matrix (index mean linear address in array)

|  |  |
| --- | --- |
| Idx 0 | Idx 2 |
| Idx 1 | Idx 3 |

Result matrix (index mean linear address in array)

|  |  |
| --- | --- |
| Idx 0 | Idx 2 |
| Idx 1 | Idx 3 |

So to emulate matrix-matrix multiplication with matrix-vector multiplication. We just divide the matrix into vector like this. B : dense matrix , C : result matrix

for (int i = 0; i < M ; i++)

{

CHECK\_CUSPARSE(cusparseDhybmv(handle,CUSPARSE\_OPERATION\_NON\_TRANSPOSE,&alpha,Adescr,hybA,&dB[i\*N],&beta,&dC[i\*M]));

}

Step 11. Get a result vector back to the host

1) cudaMemcpy

Step 12. Free the memory and matrix descriptor and resources

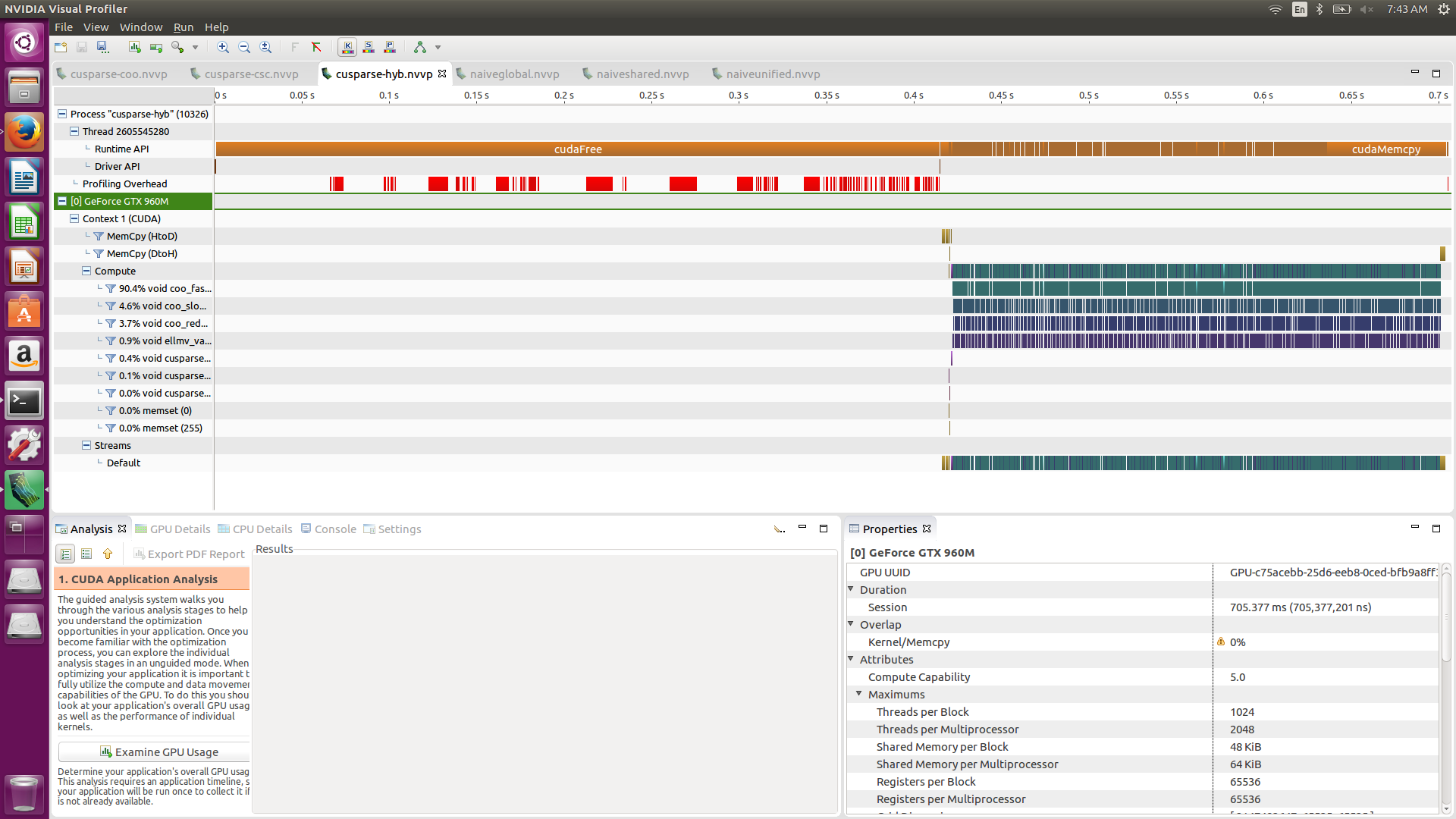
1) cudaFree

2) cusparseDestroyHybMa

3) cusparseDestroyMatDescr

4) cusparseDestroy

[screen shot]



[source code]

#include "common.h"

#include <stdio.h>

#include <stdlib.h>

#include <cusparse\_v2.h>

#include <cuda.h>

/\*

\* This is an example demonstrating usage of the cuSPARSE library to perform a

\* sparse matrix-vector multiplication on randomly generated data.

\*/

/\*

\* M = # of rows

\* N = # of columns

\*/

int M = 1024;

int N = 1024;

/\*

\* Generate random dense matrix A in column-major order, while rounding some

\* elements down to zero to ensure it is sparse.

\*/

int generate\_random\_dense\_matrix(int M, int N, double \*\*outA)

{

int i, j;

double rMax = (double)RAND\_MAX;

double \*A = (double \*)malloc(sizeof(double) \* M \* N);

int totalNnz = 0;

for (j = 0; j < N; j++)

{

for (i = 0; i < M; i++)

{

int r = rand();

double \*curr = A + (j \* M + i);

if (r % 3 > 0)

{

\*curr = 0.0;

}

else

{

double dr = (double)r;

\*curr = (dr / rMax) \* 100.0;

}

if (\*curr != 0.0)

{

totalNnz++;

}

}

}

\*outA = A;

return totalNnz;

}

// printf the partial matrix

// return : void

// parameter : double \* M - input matrix , int nrows - number of row in matrix, int ncols - number of column in matrix, int max\_row - the max row to be printed, int max\_col - the max col to be printed

void print\_partial\_matrix(double \*M, int nrows, int ncols, int max\_row,

int max\_col)

{

int row, col;

for (row = 0; row < max\_row; row++)

{

for (col = 0; col < max\_col; col++)

{

printf("%2.2f ", M[row \* ncols + col]);

}

printf("...\n");

}

printf("...\n");

}

int main(int argc, char \*\*argv)

{

double \*A, \*dA;

double \*B, \*dB;

double \*C, \*dC;

int \*dANnzPerRow;

int totalANnz;

double alpha = 1.0;

double beta = 4.0;

cusparseHandle\_t handle = 0;

cusparseHybMat\_t hybA;

cusparseMatDescr\_t Adescr = 0;

// Generate input

srand(9384);

int trueANnz = generate\_random\_dense\_matrix(M, N, &A);

int trueBNnz = generate\_random\_dense\_matrix(N, M, &B);

C = (double \*)malloc(sizeof(double) \* M \* M);

printf("A:\n");

print\_partial\_matrix(A, M, N, 10, 10);

printf("B:\n");

print\_partial\_matrix(B, N, M, 10, 10);

// Create the cuSPARSE handle

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t \* handle

// initialize the cuSPARSE library and creates a handle on the cuSPARSE context. it allocates hardware resources necessary for accessing the GPU

CHECK\_CUSPARSE(cusparseCreate(&handle));

// Allocate device memory for vectors and the dense form of the matrix A

// return : cudaError\_t

// parameter : void\*\* devPtr, size\_t size

// allocates (size) bytes of lineare memory on the device and returns in (\*devPtr) a pointer to the allocated memory

CHECK(cudaMalloc((void \*\*)&dA, sizeof(double) \* M \* N));

CHECK(cudaMalloc((void \*\*)&dB, sizeof(double) \* N \* M));

CHECK(cudaMalloc((void \*\*)&dC, sizeof(double) \* M \* M));

CHECK(cudaMalloc((void \*\*)&dANnzPerRow, sizeof(int) \* M));

// Construct a descriptor of the matrix A

// return : cusparseStatus\_t

// parameter : cudsparseMatDescr\_t \* descrA

// the function initialize the matrix descriptor.

// It sets the fields MatrixType and IndexBase to the default values CUSPARSE\_MATRIX\_TYPE\_GENERAL and CUSPARSE\_INDEX\_BASE\_ZERO

CHECK\_CUSPARSE(cusparseCreateMatDescr(&Adescr));

// create and initialize the hybA opaque data structure

// return : cusparseStatus\_t

// parameter : cusparseHybMat\_t \* hybA

CHECK\_CUSPARSE(cusparseCreateHybMat(&hybA));

// set the MatrixType field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA, cusparseMatrixType\_t type

CHECK\_CUSPARSE(cusparseSetMatType(Adescr, CUSPARSE\_MATRIX\_TYPE\_GENERAL));

// set the IndexBase field of the matrix descriptor descrA

// return : cusparseStatus\_t

// parameter : cusparseMatDescr\_t descrA , cusparseIndexBase\_t base

CHECK\_CUSPARSE(cusparseSetMatIndexBase(Adescr, CUSPARSE\_INDEX\_BASE\_ZERO));

// Transfer the input vectors and dense matrix A to the device

// return : cudaError\_t

// parameter : void\* dst, const void\* src, size\_t count, enum cudaMemcpyKind kind

// copies (count) bytes from the memory area pointed by (src) to the memory area pointed to by (dst), where (kind) is one of enum type

CHECK(cudaMemcpy(dA, A, sizeof(double) \* M \* N, cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(dB, B, sizeof(double) \* N \* M, cudaMemcpyHostToDevice));

CHECK(cudaMemset((void\*)dC, 0, sizeof(double) \* M \* M));

// Compute the number of non-zero elements in A and compute the number of non-zero elements in A per row

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle , cusparseDirection\_t dirA, int m, int n, const cusparseMatDescr\_t descrA, const double \*A, int lda, int\* nnzPerRowColumn, int\* nnzTotalDevHostPtr

// input : handle - handle to the cuSPARSE library context

// dirA - direction that specifies whether to count nonzero elements by CUSPARSE\_DIRECTION\_ROW or by CUSPARSE\_DIRECTION\_COLUMN

// m - number of rows of matrix A

// n - number of columns of matrix A

// descrA - the descriptor of matrix A

// A - array of dimensions (lda,n)

// lda - leading dimension of dense array A

// output: nnzPerRowColumn - array of size m

CHECK\_CUSPARSE(cusparseDnnz(handle, CUSPARSE\_DIRECTION\_ROW, M, N, Adescr,

dA, M, dANnzPerRow, &totalANnz));

if (totalANnz != trueANnz)

{

fprintf(stderr, "Difference detected between cuSPARSE NNZ and true "

"value: expected %d but got %d\n", trueANnz, totalANnz);

return 1;

}

// convert matrix A in dense format into a sparse matrix in HYB format.

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle, int m, int n, const cusparseMatDescr\_t descrA, const double \*A, int lda, const int \*nnzperRow, cusparseHybMat\_t hybA, int userEllWidth, cusparseHybPartition\_t partitionType

CHECK\_CUSPARSE(cusparseDdense2hyb(handle,M,N,Adescr,dA,M,dANnzPerRow,hybA,0,CUSPARSE\_HYB\_PARTITION\_AUTO));

// Perform matrix-vector multiplication with the hyb-formatted matrix A

// there is no direct way to calculate hyb matrix multiplication so, we use serveral matrix-vector multiplication to get a result.

for (int i = 0; i < M ; i++)

{

// perform the matrix-vector operation

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle, cusparseOperation\_t transA, const double \* alpha, const cusparseMatDescr\_t descrA, const cusparseHybMat\_t hybA, const double \*x, const double \*beta, double \*y

CHECK\_CUSPARSE(cusparseDhybmv(handle,CUSPARSE\_OPERATION\_NON\_TRANSPOSE,&alpha,Adescr,hybA,&dB[i\*N],&beta,&dC[i\*M]));

}

// Copy the result vector back to the host

CHECK(cudaMemcpy(C, dC, sizeof(double) \* M \* M, cudaMemcpyDeviceToHost));

// print the result partial matrix

printf("C:\n");

print\_partial\_matrix(C, M, M,10,10);

// free the memory

free(A);

free(B);

free(C);

CHECK(cudaFree(dA));

CHECK(cudaFree(dB));

CHECK(cudaFree(dC));

CHECK(cudaFree(dANnzPerRow));

// destroys and releases any memory required by the hybA structure.

// return : cusparseStatus\_t

// parameter : cusparseHybMat\_t hybA

CHECK\_CUSPARSE(cusparseDestroyHybMat(hybA));

// releases the memory allocated for the matrix descriptor

// return : cusparseStatus\_t

// parameter : cusparseDescr\_t descrA

CHECK\_CUSPARSE(cusparseDestroyMatDescr(Adescr));

// releases CPU-side resources used by the cuSPARSE library.

// return : cusparseStatus\_t

// parameter : cusparseHandle\_t handle

CHECK\_CUSPARSE(cusparseDestroy(handle));

return 0;

}

4. native multiplication compare

The goal of sparse matrix format is to reduce the memory space. So the running time with sparse matrix is longer than dense matrix. You can see the fact in the below chart.

coo and csc format run the same kernel function cusparseDcsrmm, but csc format take more time than coo format. cause they perform matrix transpose operation.

Hyb format is the longest format in this example. Cause they consist of two format ELL and COO format. so there is more calculation to support two format.

|  |  |
| --- | --- |
| Type | Running time |
| Sparse coo format | 457.927ms |
| Sparse csc format | 565.042ms |
| Sparse hyb format | 705.377ms |
| Native with Global memory | 237.791ms |
| Native with shared memory | 322.909ms |
| Native with unified memory | 419.574ms |