Homework4

20173155 jinkwon kim

Problem1

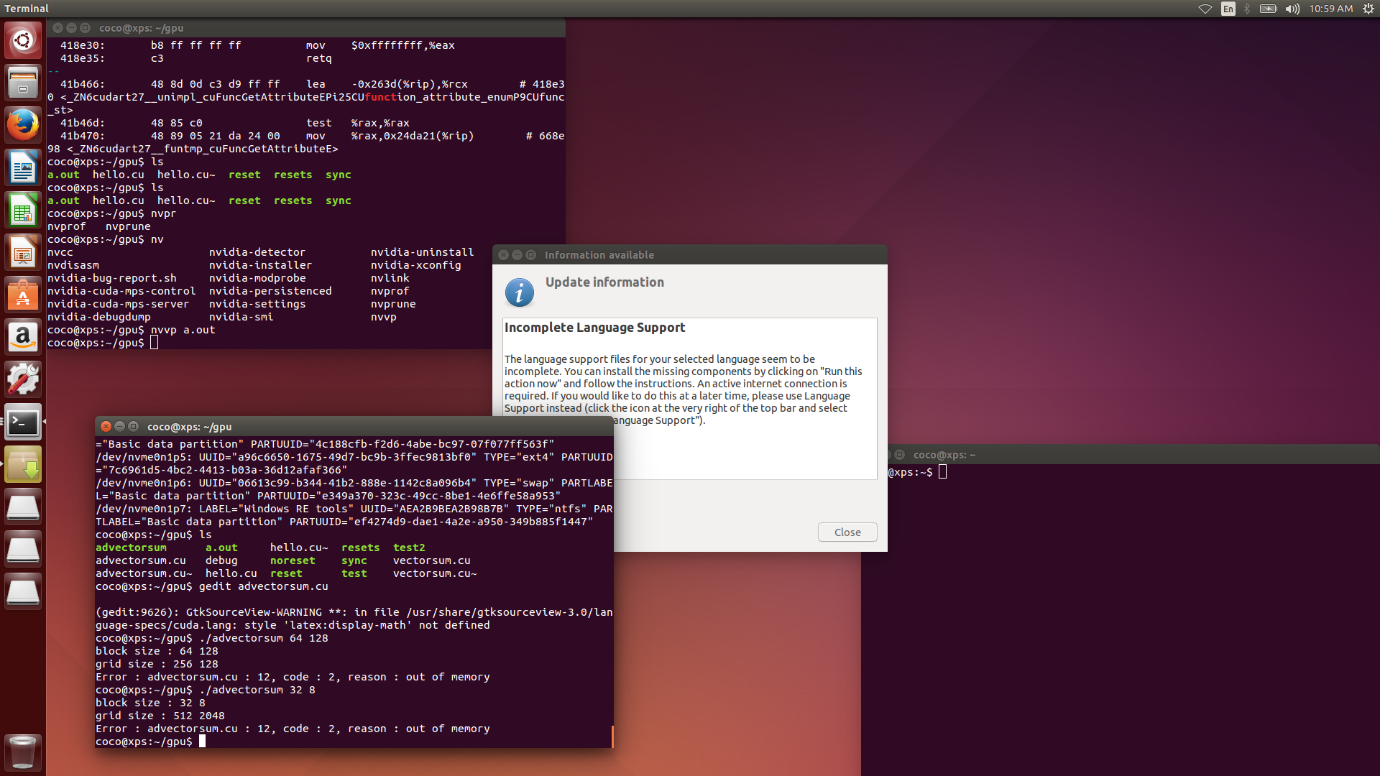
Overlapping using Depth\_first scheduling

In homework1, we develop the simple parallel addition. In this homework, we develop the parallel addition with stream.

First, I show you the nvvp result. And then discuss the result

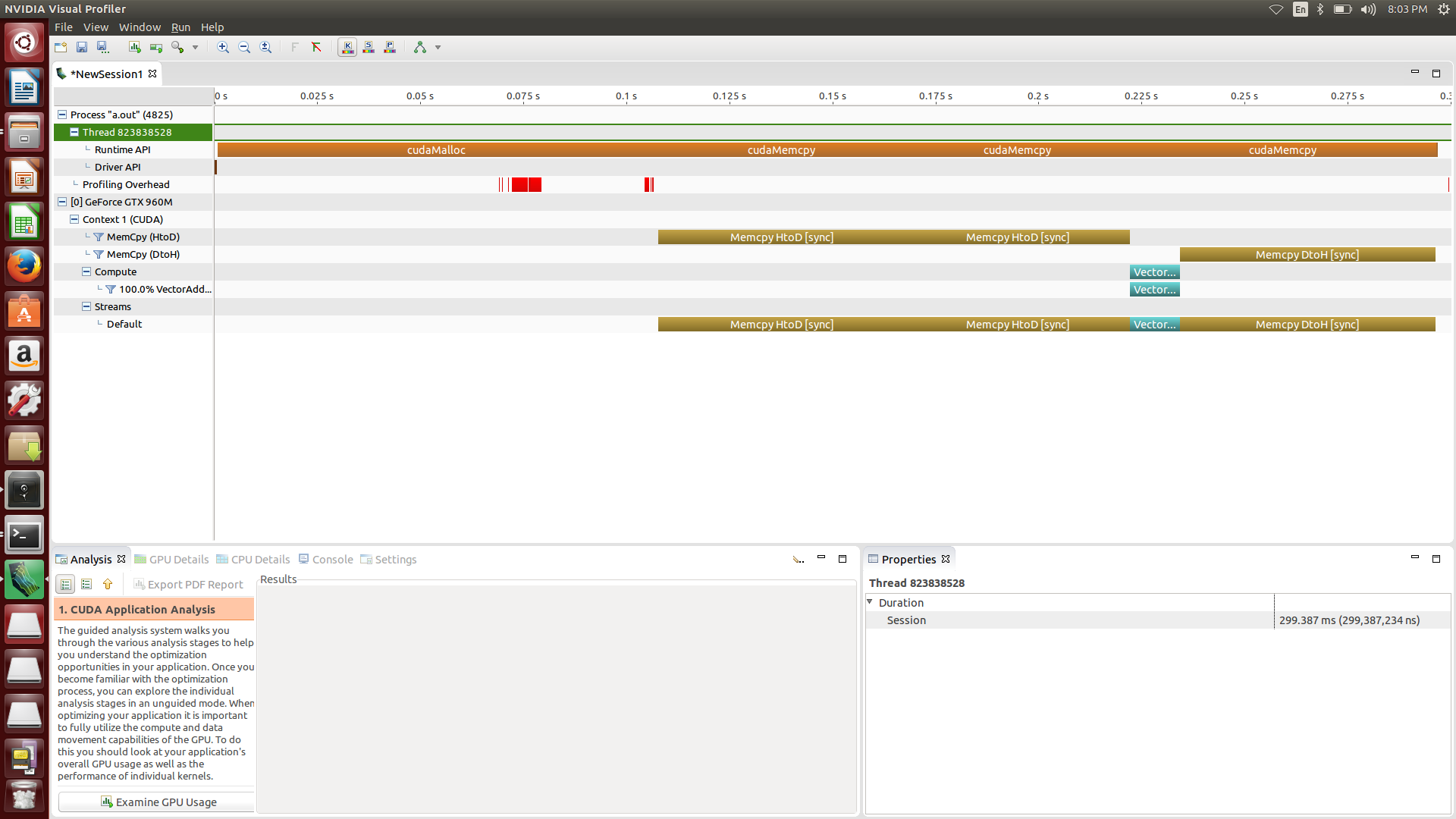
To compare the result, I set the nx, ny = 8192, block size = 32,32 (I reduce the problem size cause the memory limit. I told that in homework 1)

This is the reason I reduce the problem size.

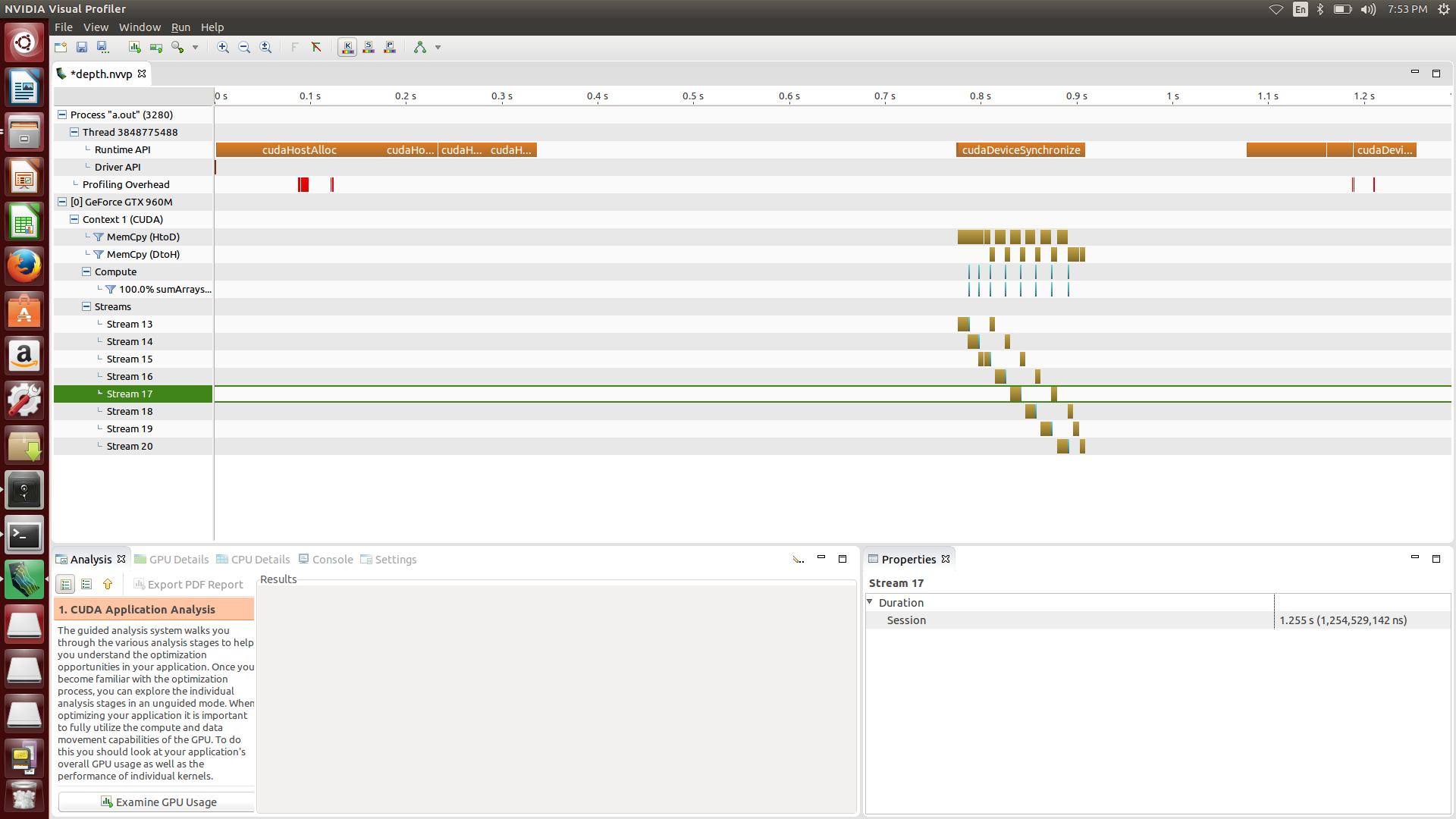


[screen shot]

Homework1



Homework4



As you can see in homework1 screenshot, first you copy the data from host to device, second launch the kernel, third copy the data from device to host.

But in homework4, with stream, we divide the problem to get a more parallelism. We use the depth-first search scheduling. my gpu supports hyper-q, so there is no false-dependency.

my gpu (gtx 960m (this is notebook version), it isn’t duplex pcie bus, so the data transfer between host and device can’t be overlapped. You can see the result in screenshot.

We just overlap the kernel and data transfer. So we can’t get a better result. (data transfer need some setup time. And we can’t overlap the data transfer, it reduce the performance)

Homework1 session duration : 299.387ms

Homework4 session duration : 1.255s

[source code]

#include "common.h"

#include <stdio.h>

#include <cuda\_runtime.h>

// number of stream in the program

#define NSTREAM 8

// host program to calculate the correct answer

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix)

// int N (size of the matrix)

void sumArraysOnHost(int \*A, int \*B, int \*C, const int N)

{

// calculate the answer

for (int idx = 0; idx < N; idx++)

C[idx] = A[idx] + B[idx];

}

// matrix addition kernel function

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix, C)

// int N (size of the matrix )

\_\_global\_\_ void sumArrays(int \*A, int \*B, int \*C, const int N)

{

// calculate the unique threadId

// unique threadId can be used to determine the position of addition

int ix = blockIdx.x \* blockDim.x + threadIdx.x;

int iy = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned int idx = iy \* blockDim.x \* gridDim.x + ix;

if (idx < N)

{

C[idx] = A[idx] + B[idx];

}

}

// host and kernel result compare function

// parameter : int\* hostRef (host result)

// int\* gpuRef (kernel result)

// int N (size of the matrix)

void checkResult(int \*hostRef,int \*gpuRef, const int N)

{

bool match = 1;

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

{

// if the result is not same

if (hostRef[i] != gpuRef[i])

{

match = 0;

printf("Arrays do not match!\n");

printf("host %d gpu %d at %d\n", hostRef[i], gpuRef[i], i);

break;

}

}

// if the result is same

if (match) printf("Arrays match.\n\n");

}

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

{

int dev = 0;

cudaDeviceProp deviceProp;

// get device information

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

printf("> Using Device %d: %s\n", dev, deviceProp.name);

// return : cudaError\_t

// parameter : int device (device on which the active host thread should execute the device code)

// records device as the device on which the active host thread executes the device code

CHECK(cudaSetDevice(dev));

// set up max connectioin

setenv ("CUDA\_DEVICE\_MAX\_CONNECTIONS", "4", 1);

int nx = 8192;

int ny = 8192;

int dimx = 32;

int dimy = 32;

// each stream subset problem size

int subset = nx\*ny/ NSTREAM;

// kernel configuration

// we just slice the x-axis to divide the problem

dim3 block(dimx,dimy);

dim3 grid((nx/NSTREAM+block.x-1)/block.x,(ny+block.y-1)/block.y);

// malloc pinned host memory for async memcpy

// return : cudaError\_t

// parameter : void\*\* pHost, size\_t size , unsigned int flags

// allocates (size) bytes of host memory that is page-locked and accessible to the device

int \*h\_A, \*h\_B, \*hostRef, \*gpuRef;

CHECK(cudaHostAlloc((void\*\*)&h\_A, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&h\_B, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&gpuRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&hostRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

// init the data

for (int i = 0; i < (nx\*ny); i++)

{

h\_A[i] = i;

h\_B[i] = i;

hostRef[i] = 0;

gpuRef[i] = 0;

}

// add vector at host side for result checks

sumArraysOnHost(h\_A, h\_B, hostRef, nx\*ny);

// malloc device global memory

int \*d\_A, \*d\_B, \*d\_C;

// 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((int\*\*)&d\_A, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_B, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_C, nx\*ny\*sizeof(int)));

// stream declare

cudaStream\_t stream[NSTREAM];

// create a new asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t\* pStream

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

{

CHECK(cudaStreamCreate(&stream[i]));

}

// initiate all work on the device asynchronously in depth-first order

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

{

int ioffset = i \* subset;

CHECK(cudaMemcpyAsync(&d\_A[ioffset], &h\_A[ioffset], subset\*sizeof(int),

cudaMemcpyHostToDevice, stream[i]));

CHECK(cudaMemcpyAsync(&d\_B[ioffset], &h\_B[ioffset], subset\*sizeof(int),

cudaMemcpyHostToDevice, stream[i]));

sumArrays<<<grid, block, 0, stream[i]>>>(&d\_A[ioffset], &d\_B[ioffset],

&d\_C[ioffset], subset);

CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d\_C[ioffset], subset\*sizeof(int),

cudaMemcpyDeviceToHost, stream[i]));

}

// return : cudaError\_t

// parameter : void

// blocks until the device has completed all preceding requsted tasks.

CHECK(cudaDeviceSynchronize());

// check device results

checkResult(hostRef, gpuRef, nx\*ny);

// free device global memory

// return : cudaError\_t

// parameter : void\* devPtr

// free the memory space pointed to by devPtr

CHECK(cudaFree(d\_A));

CHECK(cudaFree(d\_B));

CHECK(cudaFree(d\_C));

// free page-locked memory

// return : cudaError\_t

// parameter : void\* ptr

CHECK(cudaFreeHost(h\_A));

CHECK(cudaFreeHost(h\_B));

CHECK(cudaFreeHost(hostRef));

CHECK(cudaFreeHost(gpuRef));

// destroys and cleans up an asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t stream

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

{

CHECK(cudaStreamDestroy(stream[i]));

}

// reset device

// return : cudaError\_t

// parameter : void

// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return(0);

}

Problem2

Overlapping using breath\_first scheduling

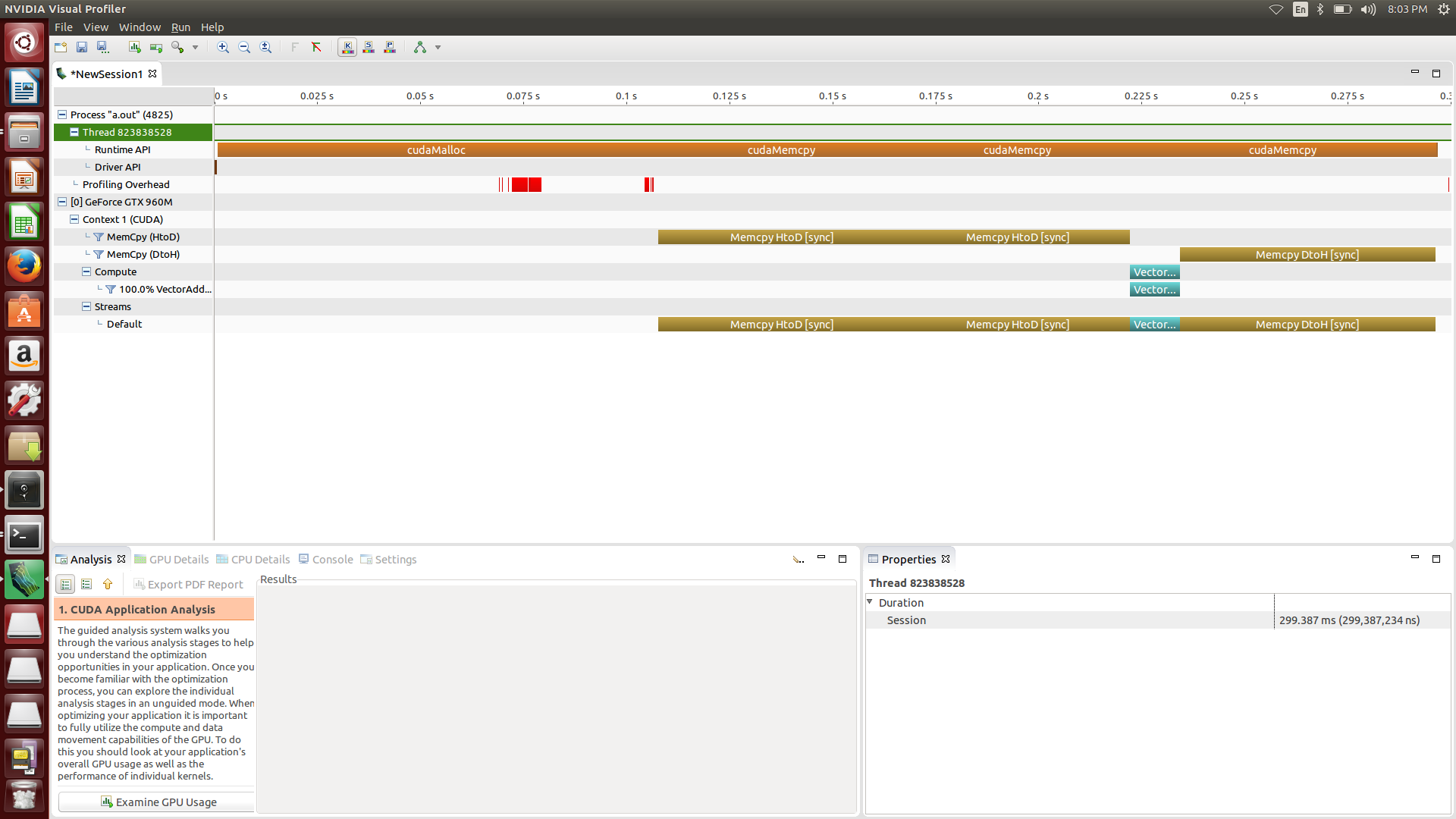
As similar in problem1, I compare the breath\_first scheduling with homework1

As same in problem1, I reduce the problem nx,ny = 8192, block size = 32,32 (same reason in problem1)

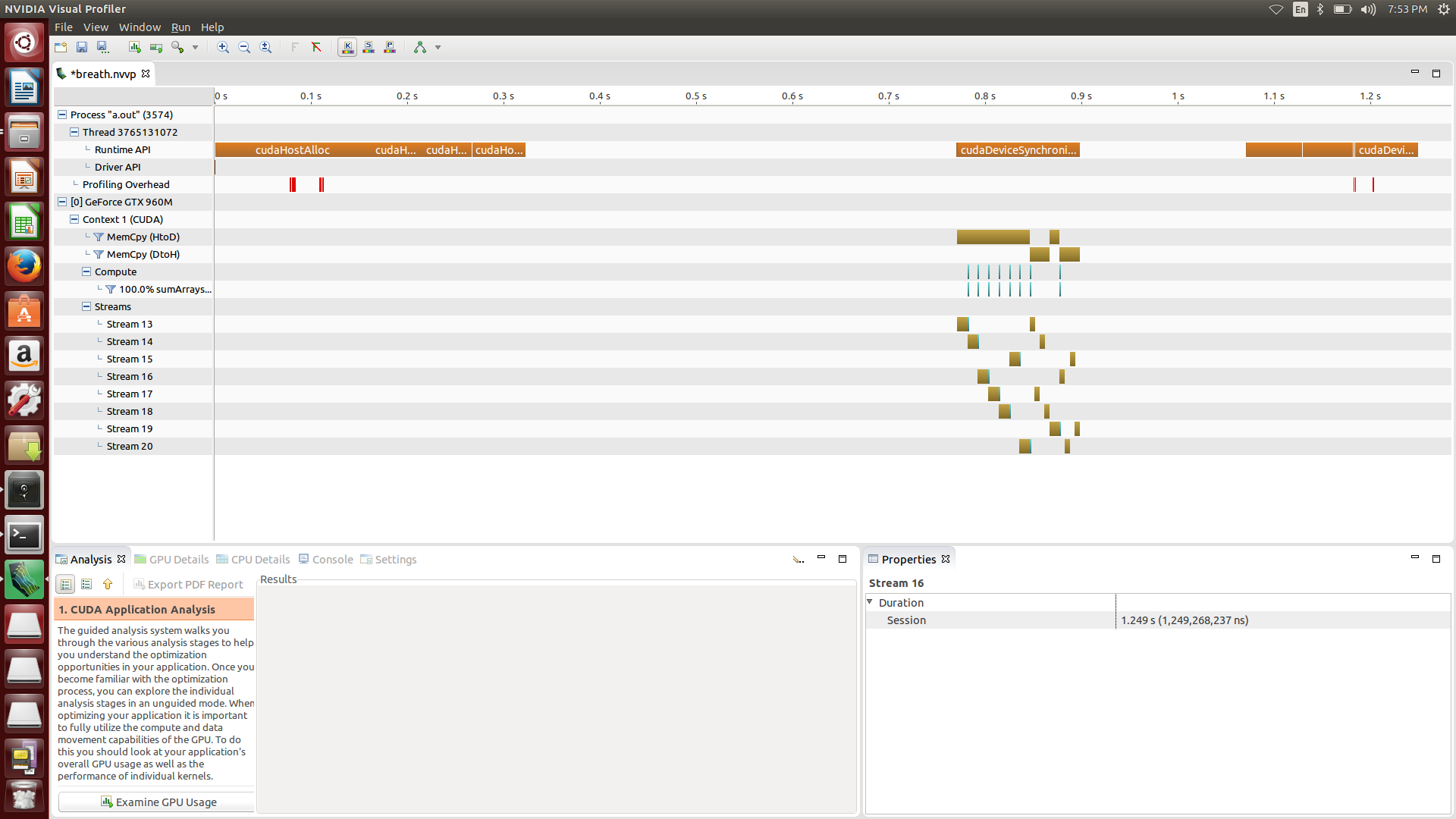
First, I show you the nvvp result. And then discuss the result

[screen shot]

Homework1



Homework4



As you can see in homework1 screenshot, first you copy the data from host to device, second launch the kernel, third copy the data from device to host.

But in homework4, with stream, we divide the problem to get a more parallelism. We use the breath-first search scheduling. In breath first search, when you use the stream lower than the device connection, false-dependency still exists. But in this problem kernel time is short, you can’t see the false-dependency.

my gpu (gtx 960m (this is notebook version), it isn’t duplex pcie bus, so the data transfer between host and device can’t be overlapped. You can see the result in screenshot.

We just overlap the kernel and data transfer. So we can’t get a better result. (data transfer need some setup time. And we can’t overlap the data transfer, it reduce the performance)

Homework1 session duration : 299.387ms

Homework4 session duration : 1.249s

[source code]

#include "common.h"

#include <stdio.h>

#include <cuda\_runtime.h>

// number of stream in the program

#define NSTREAM 8

// host program to calculate the correct answer

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix)

// int N (size of the matrix)

void sumArraysOnHost(int \*A, int \*B, int \*C, const int N)

{

// calculate the answer

for (int idx = 0; idx < N; idx++)

C[idx] = A[idx] + B[idx];

}

// matrix addition kernel function

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix, C)

// int N (size of the matrix )

\_\_global\_\_ void sumArrays(int \*A, int \*B, int \*C, const int N)

{

// calculate the unique threadId

// unique threadId can be used to determine the position of addition

int ix = blockIdx.x \* blockDim.x + threadIdx.x;

int iy = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned int idx = iy \* blockDim.x \* gridDim.x + ix;

if (idx < N)

{

C[idx] = A[idx] + B[idx];

}

}

// host and kernel result compare function

// parameter : int\* hostRef (host result)

// int\* gpuRef (kernel result)

// int N (size of the matrix)

void checkResult(int \*hostRef,int \*gpuRef, const int N)

{

bool match = 1;

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

{

// if the result is not same

if (hostRef[i] != gpuRef[i])

{

match = 0;

printf("Arrays do not match!\n");

printf("host %d gpu %d at %d\n", hostRef[i], gpuRef[i], i);

break;

}

}

// if the result is same

if (match) printf("Arrays match.\n\n");

}

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

{

int dev = 0;

cudaDeviceProp deviceProp;

// get device information

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

printf("> Using Device %d: %s\n", dev, deviceProp.name);

// return : cudaError\_t

// parameter : int device (device on which the active host thread should execute the device code)

// records device as the device on which the active host thread executes the device code

CHECK(cudaSetDevice(dev));

// set up max connectioin

setenv ("CUDA\_DEVICE\_MAX\_CONNECTIONS", "4", 1);

int nx = 8192;

int ny = 8192;

int dimx = 32;

int dimy = 32;

// each stream subset problem size

int subset = nx\*ny/ NSTREAM;

// kernel configuration

// we just slice the x-axis to divide the problem

dim3 block(dimx,dimy);

dim3 grid((nx/NSTREAM+block.x-1)/block.x,(ny+block.y-1)/block.y);

// malloc pinned host memory for async memcpy

// return : cudaError\_t

// parameter : void\*\* pHost, size\_t size , unsigned int flags

// allocates (size) bytes of host memory that is page-locked and accessible to the device

int \*h\_A, \*h\_B, \*hostRef, \*gpuRef;

CHECK(cudaHostAlloc((void\*\*)&h\_A, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&h\_B, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&gpuRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&hostRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

// init the data

for (int i = 0; i < (nx\*ny); i++)

{

h\_A[i] = i;

h\_B[i] = i;

hostRef[i] = 0;

gpuRef[i] = 0;

}

// add vector at host side for result checks

sumArraysOnHost(h\_A, h\_B, hostRef, nx\*ny);

// malloc device global memory

int \*d\_A, \*d\_B, \*d\_C;

// 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((int\*\*)&d\_A, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_B, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_C, nx\*ny\*sizeof(int)));

// stream declare

cudaStream\_t stream[NSTREAM];

// create a new asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t\* pStream

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

{

CHECK(cudaStreamCreate(&stream[i]));

}

// breath first search step 1

// copy the A,B matrix to the device

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

{

int ioffset = i \* subset;

CHECK(cudaMemcpyAsync(&d\_A[ioffset], &h\_A[ioffset], subset\*sizeof(int),

cudaMemcpyHostToDevice, stream[i]));

CHECK(cudaMemcpyAsync(&d\_B[ioffset], &h\_B[ioffset], subset\*sizeof(int),

cudaMemcpyHostToDevice, stream[i]));

}

// breath first search step 2

// kernel launch

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

{

int ioffset = i \* subset;

sumArrays<<<grid, block, 0, stream[i]>>>(&d\_A[ioffset], &d\_B[ioffset],

&d\_C[ioffset], subset);

}

// breath first search step 3

// get a result matrix C from the device

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

{

int ioffset = i \* subset;

CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d\_C[ioffset], subset\*sizeof(int),

cudaMemcpyDeviceToHost, stream[i]));

}

// return : cudaError\_t

// parameter : void

// blocks until the device has completed all preceding requsted tasks.

CHECK(cudaDeviceSynchronize());

// check device results

checkResult(hostRef, gpuRef, nx\*ny);

// free device global memory

// return : cudaError\_t

// parameter : void\* devPtr

// free the memory space pointed to by devPtr

CHECK(cudaFree(d\_A));

CHECK(cudaFree(d\_B));

CHECK(cudaFree(d\_C));

// free page-locked memory

// return : cudaError\_t

// parameter : void\* ptr

CHECK(cudaFreeHost(h\_A));

CHECK(cudaFreeHost(h\_B));

CHECK(cudaFreeHost(hostRef));

CHECK(cudaFreeHost(gpuRef));

// destroys and cleans up an asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t stream

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

{

CHECK(cudaStreamDestroy(stream[i]));

}

// reset device

// return : cudaError\_t

// parameter : void

// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return(0);

}

Problem3

**Matrix Multiplication Using Breath\_First Scheduling**

In homework3, we develop matrix multiplication with global memory, shared memory, unified memory.

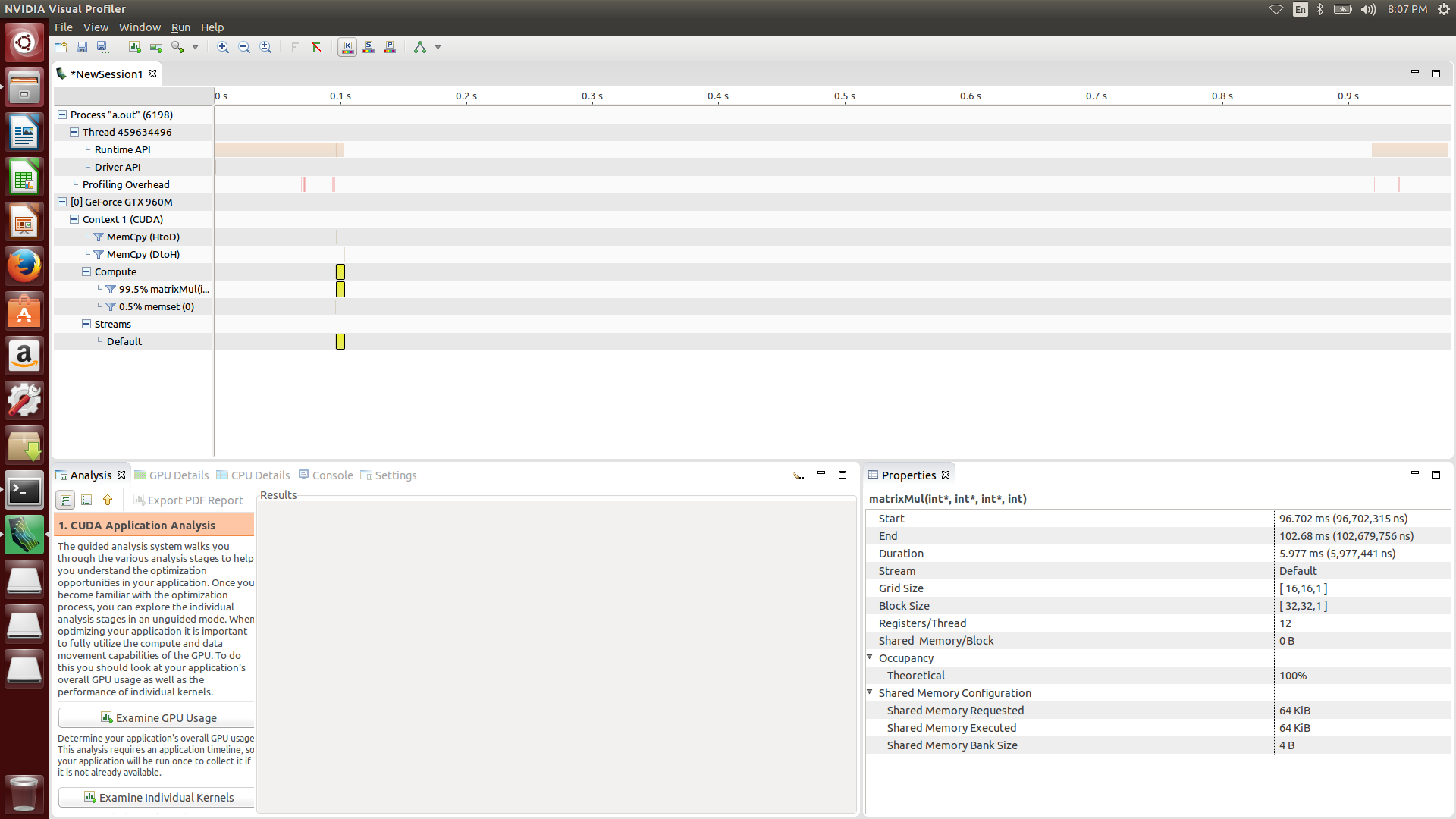
First I show you the screen shot, and then I will explain the reason.

Cause memory limit, nx = 512, ny = 512, block size = 32,32

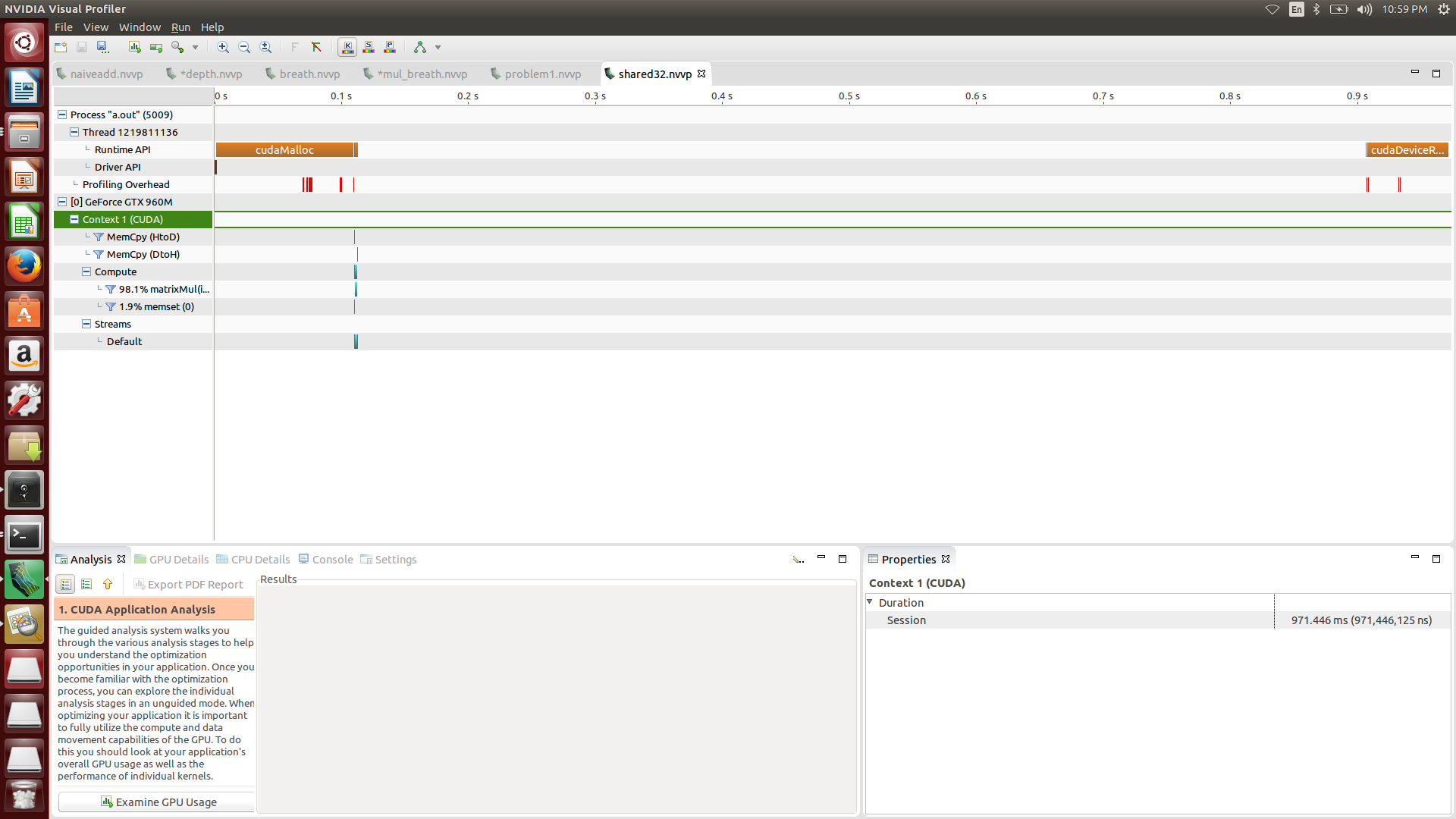
[screen shot]

Homework3

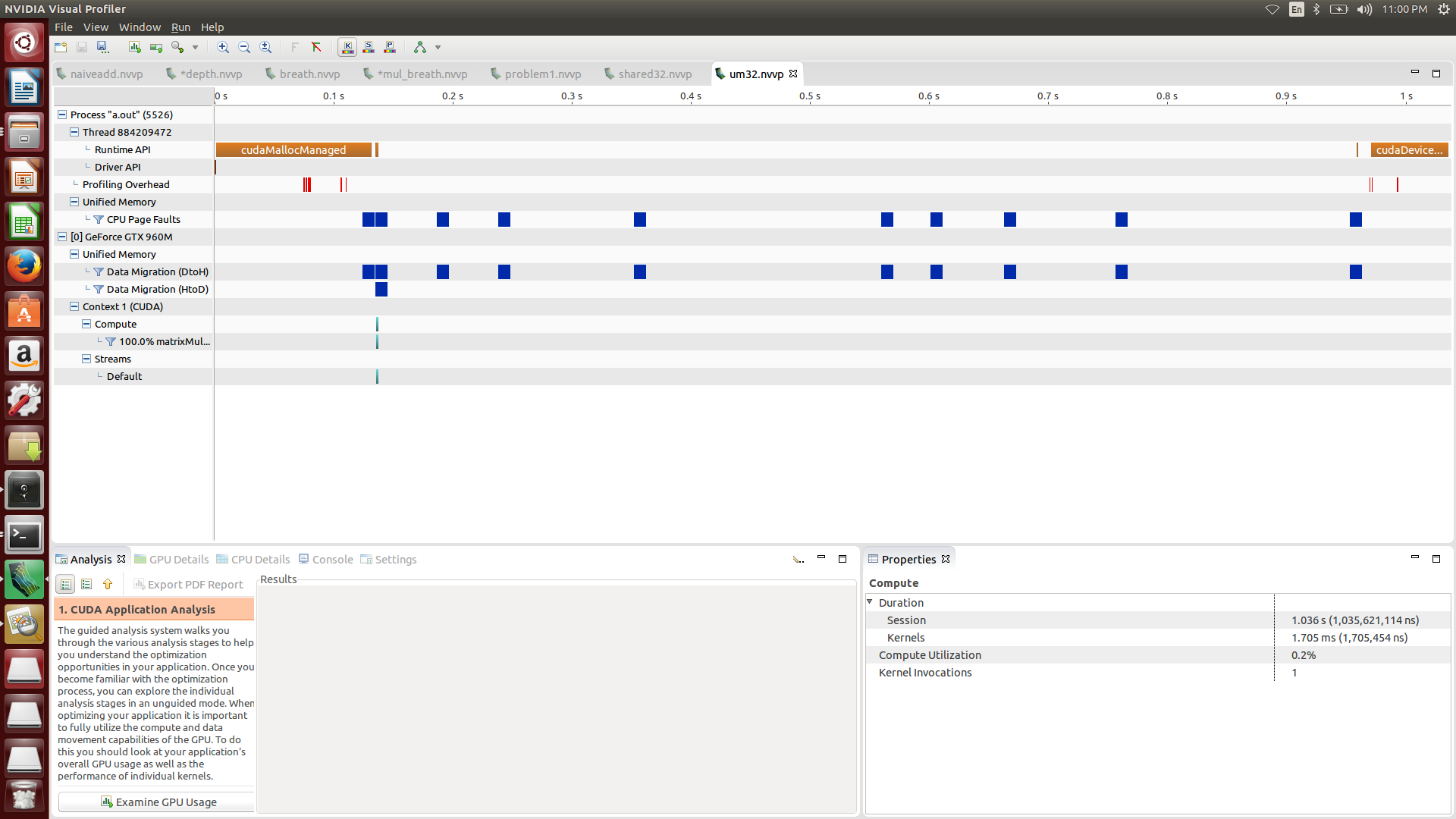
1. Global memory

****

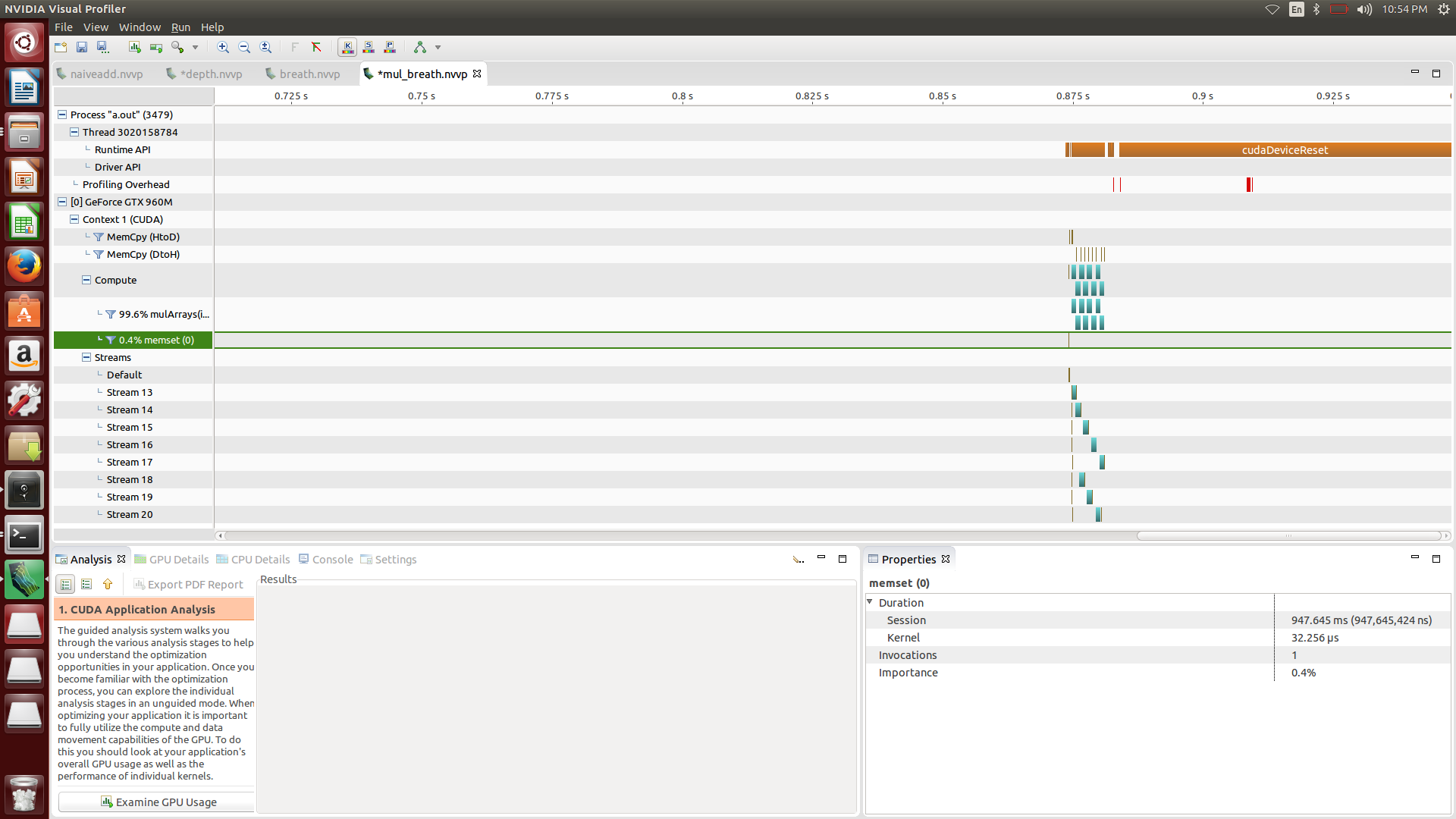
1. Shared memory



1. Unified memory



Homework4



Problem1 with global memory session duration : 987.875ms

Problem1 with shared memory session duration : 971.446ms

Problem1 with unified memory session duration : 1.036s

Problem 4 memory session : 947.645ms

In problem 1, using shared memory is the fastest one, cause shared memory access time is same as L1 cache. and the unified memory is the lowest one, cause it copy the host and device memory with prediction. so, the prediction need some time to copy the data and maybe it can be wrong. So the unified memory is the lowest one.

I told you that my gpu isn’t duplex pcie, you can see the only one direction memory copy.

And in this example you can see the kernel overlapping. Cause matrix multiplication is take some time. So the session duration is faster than problem 1.

Before I show you the source code, I explain about my implementation.

Let me assume that all the matrix is square and to simplify the size is 2\*2 and the calculation is like that AB = C

When you divide the problem with stream, the easiest way is divide the output value.

So, first I divide the output value in vertical cut(matrix C)

Matrix C

|  |  |
| --- | --- |
| 1 | 1 |
| 2 | 2 |

To calculate the matrix C, you need some part of matrix A, and total matrix B

When you get a first row of matrix C , you need first row of matrix A, and total matrix B.

When you get a second row of matrix C, you need second row of matrix A, and total matrix B.

So I first copy the total matrix B from host to device and call the cudaDeviceSynchronize()) to synchronize. And then launch the stream.

[source code]

#include "common.h"

#include <stdio.h>

#include <cuda\_runtime.h>

// number of stream in the program

#define NSTREAM 8

// host program to calculate the correct answer

// we assume that input matrix is always square

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix, C= AB)

// int nx (size of the matrix row&column)

void mulArraysOnHost(int \*A, int \*B, int \*C, const int size)

{

for (int col = 0; col < size; col++)

{

for (int row = 0; row < size; row++)

{

int outidx = col\*size + row;

for (int idx = 0; idx < size; idx++)

{

C[outidx] += A[col\*size + idx]\*B[idx\*size + row];

}

}

}

}

// matrix multiplcation kernel function

// parameter : int\* A (input matrix)

// int\* B (input matrix)

// int\* C (output matrix, C= AB)

// int nx (size of the matrix row)

// int ny (size of the matrix column)

\_\_global\_\_ void mulArrays(int \*A, int \*B, int \*C, const int nx,const int ny)

{

// calculate the unique threadId

// unique threadId can be used to determine the position of addition

int ix = blockIdx.x \* blockDim.x + threadIdx.x;

int iy = blockIdx.y \* blockDim.y + threadIdx.y;

unsigned int idx = iy \* blockDim.x \* gridDim.x + ix;

// matrix multiplication

if (ix < nx && iy < ny)

{

for (int temp = 0 ; temp < nx; temp++)

{

C[idx] += A[iy\*nx + temp]\*B[temp\*nx + ix];

}

}

}

// host and kernel result compare function

// parameter : int\* hostRef (host result)

// int\* gpuRef (kernel result)

// int N (size of the matrix)

void checkResult(int \*hostRef,int \*gpuRef, const int N)

{

bool match = 1;

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

{

// if the result is not same

if (hostRef[i] != gpuRef[i])

{

match = 0;

printf("Arrays do not match!\n");

printf("host %d gpu %d at %d\n", hostRef[i], gpuRef[i], i);

break;

}

}

// if the result is same

if (match) printf("Arrays match.\n\n");

}

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

{

int dev = 0;

cudaDeviceProp deviceProp;

// get device information

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

printf("> Using Device %d: %s\n", dev, deviceProp.name);

// return : cudaError\_t

// parameter : int device (device on which the active host thread should execute the device code)

// records device as the device on which the active host thread executes the device code

CHECK(cudaSetDevice(dev));

// set up max connectioin

setenv ("CUDA\_DEVICE\_MAX\_CONNECTIONS", "4", 1);

int nx = 512;

int ny = 512;

int dimx = 32;

int dimy = 32;

// each stream subset problem size

int subset = nx\*ny/ NSTREAM;

// kernel configuration

// we slice the y-axis to divide the problem

// to assign each stream 1/8 problem

// we should divide the output matrix 1/8 (C=AB)

// to get a 1/8 matrix, we need 1/8 A matrix and all B matrix

// ex) A (32,32) B (32,32)

// to get a C (8,32) = A(8,32) \* B(32,32)

dim3 block(dimx,dimy);

dim3 grid((nx+block.x-1)/block.x,(ny/NSTREAM+block.y-1)/block.y);

// malloc pinned host memory for async memcpy

// return : cudaError\_t

// parameter : void\*\* pHost, size\_t size , unsigned int flags

// allocates (size) bytes of host memory that is page-locked and accessible to the device

int \*h\_A, \*h\_B, \*hostRef, \*gpuRef;

CHECK(cudaHostAlloc((void\*\*)&h\_A, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&h\_B, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&gpuRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

CHECK(cudaHostAlloc((void\*\*)&hostRef, nx\*ny\*sizeof(int), cudaHostAllocDefault));

// init the data

for (int i = 0; i < (nx\*ny); i++)

{

h\_A[i] = i;

h\_B[i] = i;

hostRef[i] = 0;

gpuRef[i] = 0;

}

// add vector at host side for result checks

mulArraysOnHost(h\_A, h\_B, hostRef, nx);

// malloc device global memory

int \*d\_A, \*d\_B, \*d\_C;

// 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((int\*\*)&d\_A, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_B, nx\*ny\*sizeof(int)));

CHECK(cudaMalloc((int\*\*)&d\_C, nx\*ny\*sizeof(int)));

// first init the output matrix in device

CHECK(cudaMemset(d\_C,0,nx\*ny\*sizeof(int)));

// we copy all data from host to device in matrix B

CHECK(cudaMemcpy(d\_B,h\_B,nx\*nx\*sizeof(int),cudaMemcpyHostToDevice));

// to correct answer, we need syn the previous cudaMemcpy

// return : cudaError\_t

// parameter : void

// blocks until the device has completed all preceding requsted tasks.

CHECK(cudaDeviceSynchronize());

// stream declare

cudaStream\_t stream[NSTREAM];

// create a new asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t\* pStream

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

{

CHECK(cudaStreamCreate(&stream[i]));

}

// breath first search step 1

// copy the A (1/8 size) matrix to the device

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

{

int ioffset = i \* subset;

CHECK(cudaMemcpyAsync(&d\_A[ioffset], &h\_A[ioffset], subset\*sizeof(int),

cudaMemcpyHostToDevice, stream[i]));

}

// breath first search step 2

// kernel launch

// each kernel calculate 1/8 subproblem

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

{

int ioffset = i \* subset;

mulArrays<<<grid, block, 0, stream[i]>>>(&d\_A[ioffset], d\_B,

&d\_C[ioffset], nx,ny/NSTREAM);

}

// breath first search step 3

// get a result matrix C (1/8 size) from the device

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

{

int ioffset = i \* subset;

CHECK(cudaMemcpyAsync(&gpuRef[ioffset], &d\_C[ioffset], subset\*sizeof(int),

cudaMemcpyDeviceToHost, stream[i]));

}

// return : cudaError\_t

// parameter : void

// blocks until the device has completed all preceding requsted tasks.

CHECK(cudaDeviceSynchronize());

// check device results

checkResult(hostRef, gpuRef, nx\*ny);

// free device global memory

// return : cudaError\_t

// parameter : void\* devPtr

// free the memory space pointed to by devPtr

CHECK(cudaFree(d\_A));

CHECK(cudaFree(d\_B));

CHECK(cudaFree(d\_C));

// free host memory

// return : cudaError\_t

// parameter : void\* ptr

CHECK(cudaFreeHost(h\_A));

CHECK(cudaFreeHost(h\_B));

CHECK(cudaFreeHost(hostRef));

CHECK(cudaFreeHost(gpuRef));

// destroys and cleans up an asynchronous stream

// return : cudaError\_t

// parameter : cudaStream\_t stream

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

{

CHECK(cudaStreamDestroy(stream[i]));

}

// reset device

// return : cudaError\_t

// parameter : void

// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return(0);

}