Homework 2

20173155 김진권

1. Analyze and explain the programs ‘memTransfer.cu’ and ‘pinMemTransfer.cu’ in detail.

memTransfer.cu : first allocate host memory using malloc and allocate device memory using cudaMalloc, then copy the host memory to device memory using cudaMemcpy. but when you allocate directly host memory, it can be swapped out , so before copy the data to device, the os allocate another memory called pinned memory that can not be swapped out and copy the data to the pinned memory then copy the data in pinned memory to the device memory.

PinMemTransfer.cu : in this case, we only allocate pinned memory, so additional memory transfer doesn’t need. Just copy the data in pinned memory to the device memory, so the transfer speed is much higher than the first one.

[source code]

memTransfer.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* An example of using CUDA's memory copy API to transfer data to and from the

\* device. In this case, cudaMalloc is used to allocate memory on the GPU and

\* cudaMemcpy is used to transfer the contents of host memory to an array

\* allocated using cudaMalloc.

\*/

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

{

// set up device

int dev = 0;

// 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));

// memory size

unsigned int isize = 1 << 22;

unsigned int nbytes = isize \* sizeof(float);

// get device information

cudaDeviceProp deviceProp;

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

// print the current information

printf("%s starting at ", argv[0]);

printf("device %d: %s memory size %d nbyte %5.2fMB\n", dev,

deviceProp.name, isize, nbytes / (1024.0f \* 1024.0f));

// allocate the host memory

float \*h\_a = (float \*)malloc(nbytes);

// allocate the device memory

float \*d\_a;

CHECK(cudaMalloc((float \*\*)&d\_a, nbytes));

// initialize the host memory

for(unsigned int i = 0; i < isize; i++) h\_a[i] = 0.5f;

// transfer data from the host to the device

CHECK(cudaMemcpy(d\_a, h\_a, nbytes, cudaMemcpyHostToDevice));

// transfer data from the device to the host

CHECK(cudaMemcpy(h\_a, d\_a, nbytes, cudaMemcpyDeviceToHost));

// free memory

CHECK(cudaFree(d\_a));

free(h\_a);

// reset device

CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

}

pinMemTransfer.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* An example of using CUDA's memory copy API to transfer data to and from the

\* device. In this case, cudaMalloc is used to allocate memory on the GPU and

\* cudaMemcpy is used to transfer the contents of host memory to an array

\* allocated using cudaMalloc. Host memory is allocated using cudaMallocHost to

\* create a page-locked host array.

\*/

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

{

// set up device

int dev = 0;

// 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));

// memory size

unsigned int isize = 1 << 22;

unsigned int nbytes = isize \* sizeof(float);

// get device information

cudaDeviceProp deviceProp;

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

// whether the device can map host memory into the CUDA address

if (!deviceProp.canMapHostMemory)

{

printf("Device %d does not support mapping CPU host memory!\n", dev);

CHECK(cudaDeviceReset());

exit(EXIT\_SUCCESS);

}

// print the current information

printf("%s starting at ", argv[0]);

printf("device %d: %s memory size %d nbyte %5.2fMB canMap %d\n", dev,

deviceProp.name, isize, nbytes / (1024.0f \* 1024.0f),

deviceProp.canMapHostMemory);

// allocate pinned host memory

float \*h\_a;

CHECK(cudaMallocHost ((float \*\*)&h\_a, nbytes));

// allocate device memory

float \*d\_a;

CHECK(cudaMalloc((float \*\*)&d\_a, nbytes));

// initialize host memory

memset(h\_a, 0, nbytes);

for (int i = 0; i < isize; i++) h\_a[i] = 100.10f;

// transfer data from the host to the device

CHECK(cudaMemcpy(d\_a, h\_a, nbytes, cudaMemcpyHostToDevice));

// transfer data from the device to the host

CHECK(cudaMemcpy(h\_a, d\_a, nbytes, cudaMemcpyDeviceToHost));

// free memory

CHECK(cudaFree(d\_a));

CHECK(cudaFreeHost(h\_a));

// reset device

CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

}

2. Compare the performance of the pinned and pageable memory copies in memTransfer.cu and pinMemTransfer.cu. Use ‘nvprof’ command and measure the time for Pageable HtoD, Pageable DtoH, Pinned HtoD and Pinned DtoH for the transfer memory sizes 2M,4M,8M,16M, 32M, 64M, 128MBytes. Download the memTransfer.cu and pinMemTransfer.cu program from the ~EE817 site and you have to modify them to get the transfer memory size as a command line argument.

As I mentioned before, pinned memory doesn’t need to allocate additional memory, so the transfer speed is much faster than non-pinned memory. You can see the difference between them in the below chart.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Memory size | Pageable HtoD | Pageable DtoH | Pinned HtoD | Pinned DtoH |
| 2M | 340.06us | 337.57us | 329.02us | 326.82us |
| 4M | 682.69us | 898.85us | 643.20us | 651.58us |
| 8M | 1.4691ms | 1.8123ms | 1.2891ms | 1.2775ms |
| 16M | 3.1991ms | 3.5318ms | 2.5711ms | 2.5643ms |
| 32M | 6.5647ms | 6.9702ms | 5.1150ms | 5.1356ms |
| 64M | 13.377ms | 14.148ms | 10.229ms | 10.267ms |
| 128M | 26.956ms | 27.570ms | 21.281ms | 22.866ms |

[source code]

memTransferVariable.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* An example of using CUDA's memory copy API to transfer data to and from the

\* device. In this case, cudaMalloc is used to allocate memory on the GPU and

\* cudaMemcpy is used to transfer the contents of host memory to an array

\* allocated using cudaMalloc.

\*/

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

{

// memory size

unsigned int isize;

unsigned int nbytes;

// get the memory size as a command line argument

if (argc == 2)

{

// its default size is MB

nbytes = atoi(argv[1]) \* (1<<20);

isize = nbytes>>2;

}

else

{

printf("specify the memory size \n");

exit(0);

}

// set up device

int dev = 0;

// 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));

// get device information

cudaDeviceProp deviceProp;

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

// print the current information

printf("%s starting at ", argv[0]);

printf("device %d: %s memory size nbyte %5.2fMB\n", dev,

deviceProp.name, nbytes / (1024.0f \* 1024.0f));

// allocate the host memory

float \*h\_a = (float \*)malloc(nbytes);

// allocate the device memory

// 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

float \*d\_a;

CHECK(cudaMalloc((float \*\*)&d\_a, nbytes));

// initialize the host memory

for(unsigned int i = 0; i < isize; i++) h\_a[i] = 0.5f;

// transfer data from the host 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(d\_a, h\_a, nbytes, cudaMemcpyHostToDevice));

// transfer data from the device to the host

// 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(h\_a, d\_a, nbytes, cudaMemcpyDeviceToHost));

// free memory

// return : cudaError\_t

// parameter : void\* devPtr

// free the memory space pointed to by devPtr

CHECK(cudaFree(d\_a));

free(h\_a);

// reset device

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

}

pinMemTransferVariable.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* An example of using CUDA's memory copy API to transfer data to and from the

\* device. In this case, cudaMalloc is used to allocate memory on the GPU and

\* cudaMemcpy is used to transfer the contents of host memory to an array

\* allocated using cudaMalloc. Host memory is allocated using cudaMallocHost to

\* create a page-locked host array.

\*/

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

{

// memory size

unsigned int isize;

unsigned int nbytes;

// get the memory size as a command line argument

if (argc == 2)

{

// its default size is MB

nbytes = atoi(argv[1]) \* (1<<20);

isize = nbytes>>2;

}

else

{

printf("specify the memory size \n");

exit(0);

}

// set up device

int dev = 0;

// 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));

// get device information

cudaDeviceProp deviceProp;

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

// whether the device can map host memory into the CUDA address

if (!deviceProp.canMapHostMemory)

{

printf("Device %d does not support mapping CPU host memory!\n", dev);

CHECK(cudaDeviceReset());

exit(EXIT\_SUCCESS);

}

// print the current information

printf("%s starting at ", argv[0]);

printf("device %d: %s memory size nbyte %5.2fMB canMap %d\n", dev,

deviceProp.name, nbytes / (1024.0f \* 1024.0f),

deviceProp.canMapHostMemory);

// allocate pinned host memory

float \*h\_a;

CHECK(cudaMallocHost ((float \*\*)&h\_a, nbytes));

// allocate device memory

float \*d\_a;

CHECK(cudaMalloc((float \*\*)&d\_a, nbytes));

// initialize host memory

memset(h\_a, 0, nbytes);

for (int i = 0; i < isize; i++) h\_a[i] = 100.10f;

// transfer data from the host to the device

CHECK(cudaMemcpy(d\_a, h\_a, nbytes, cudaMemcpyHostToDevice));

// transfer data from the device to the host

CHECK(cudaMemcpy(h\_a, d\_a, nbytes, cudaMemcpyDeviceToHost));

// free memory

CHECK(cudaFree(d\_a));

CHECK(cudaFreeHost(h\_a));

// reset device

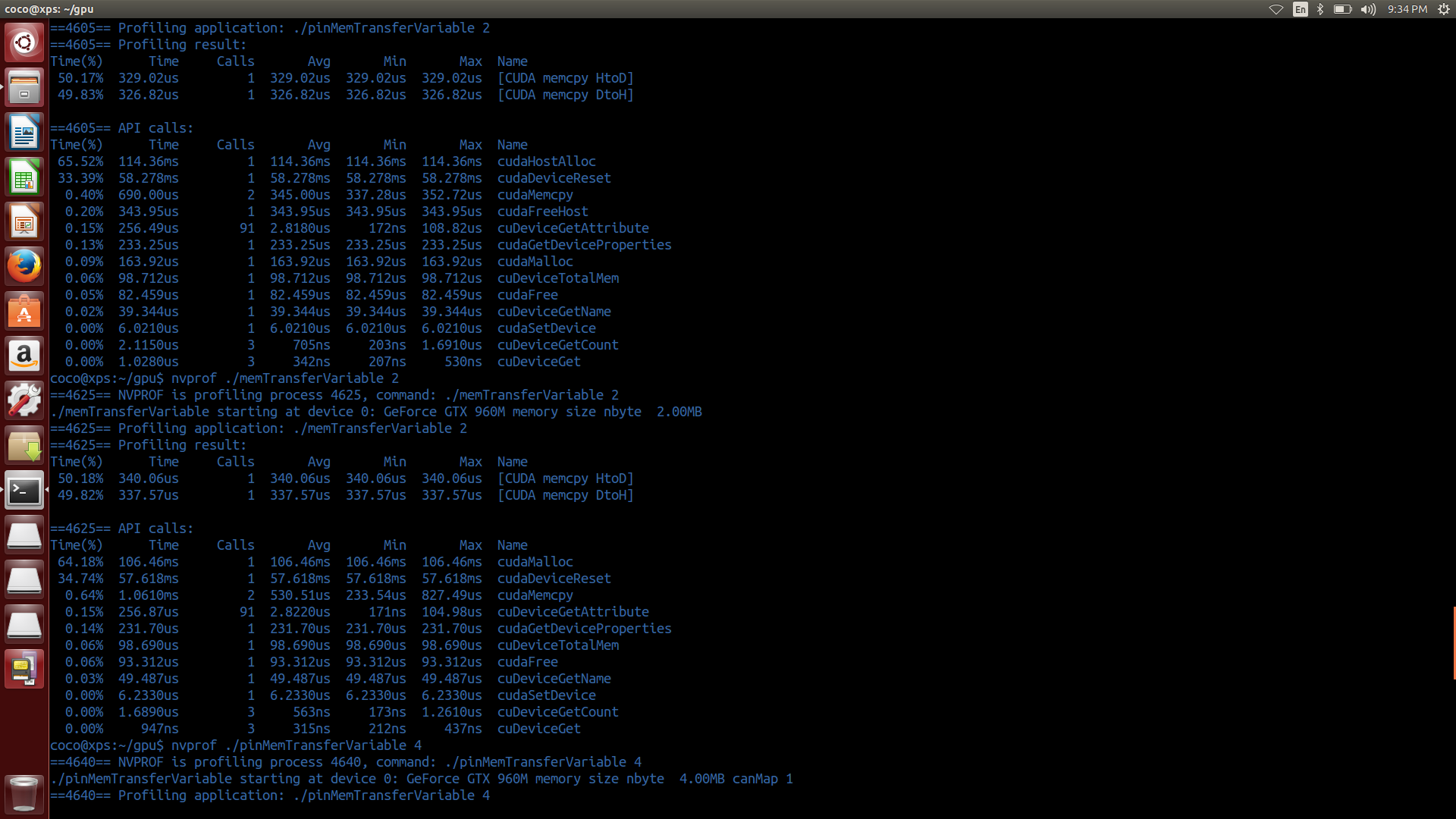
CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

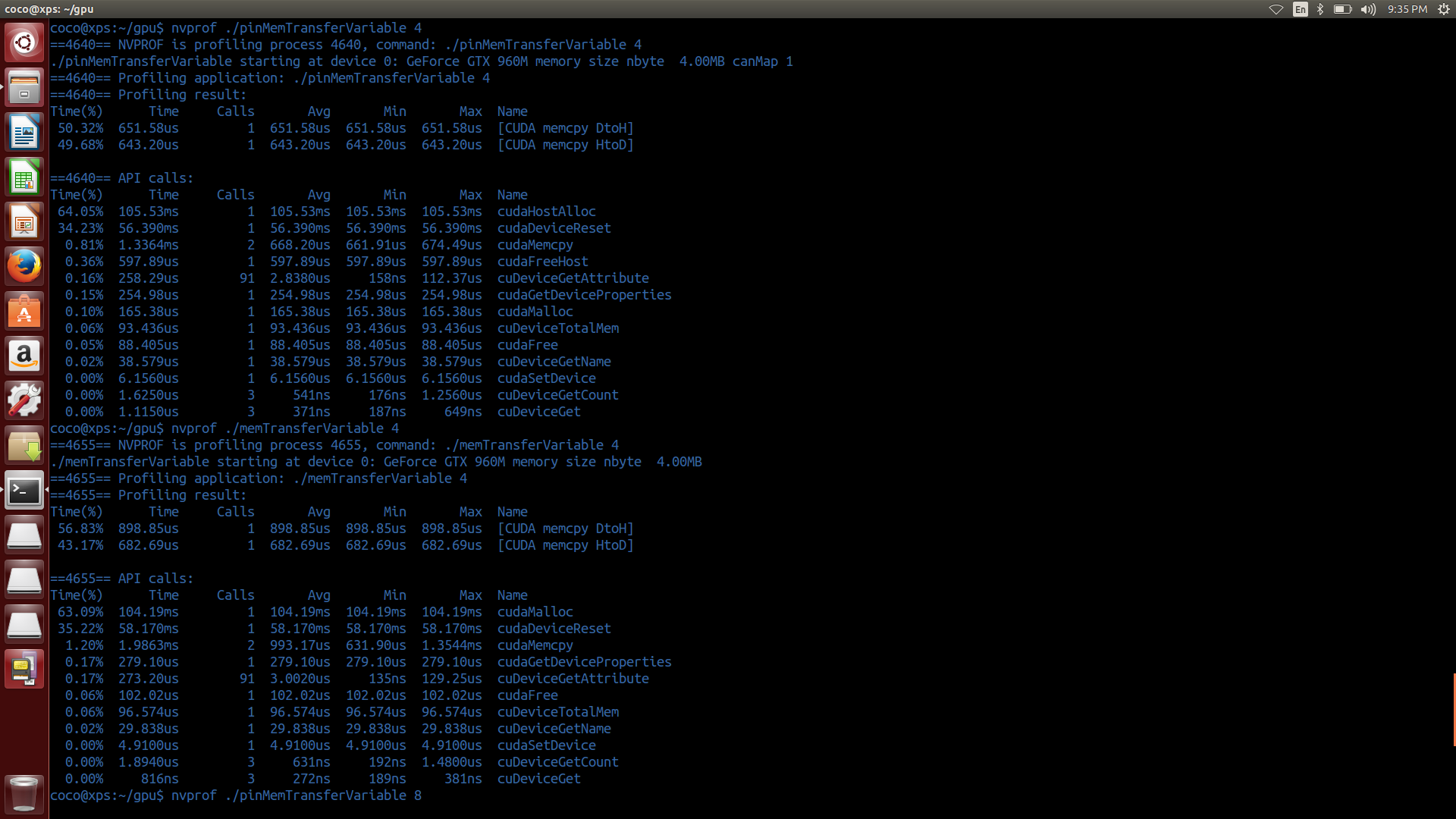
}

[screen shot]

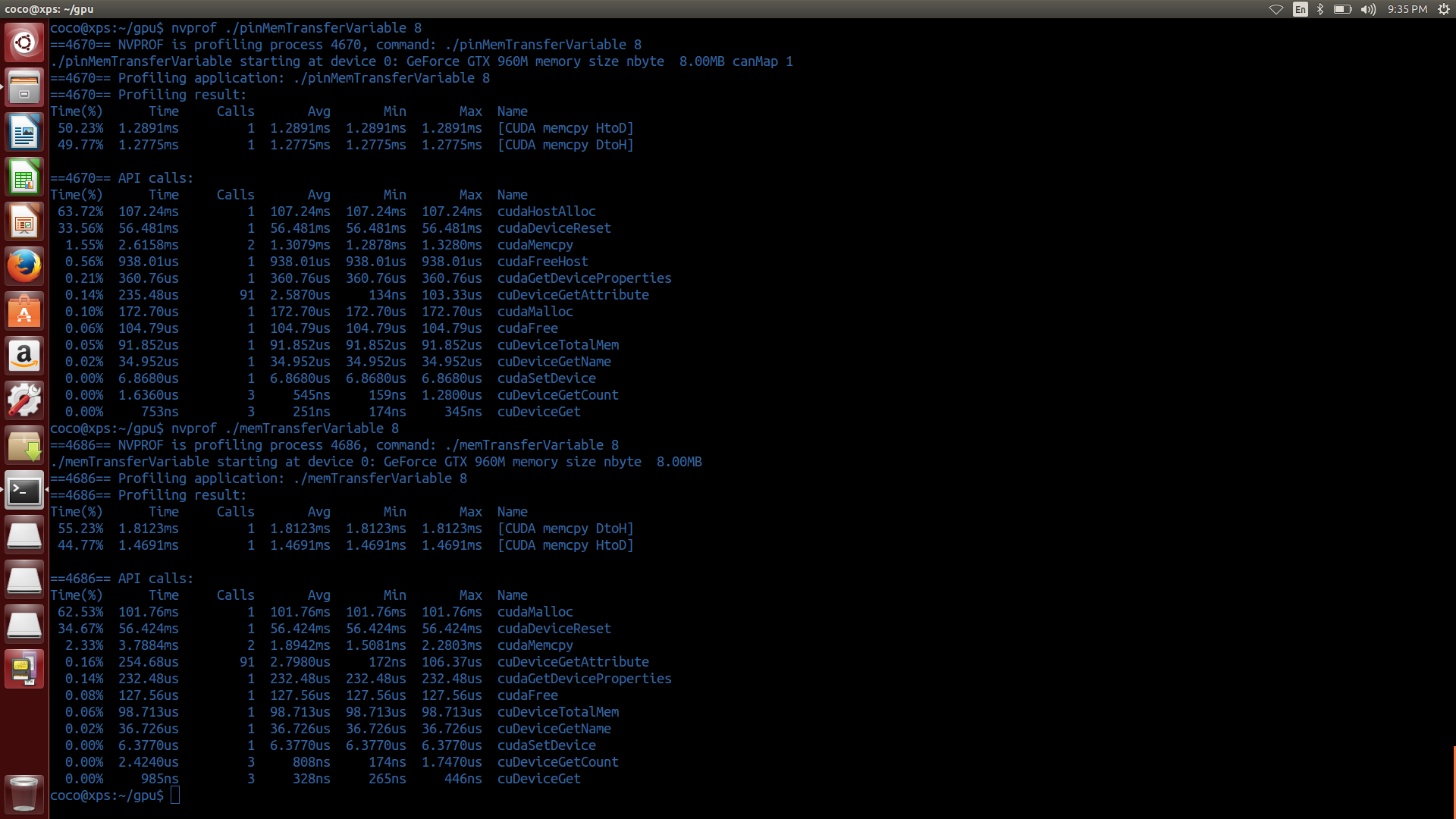
Memory size 2M



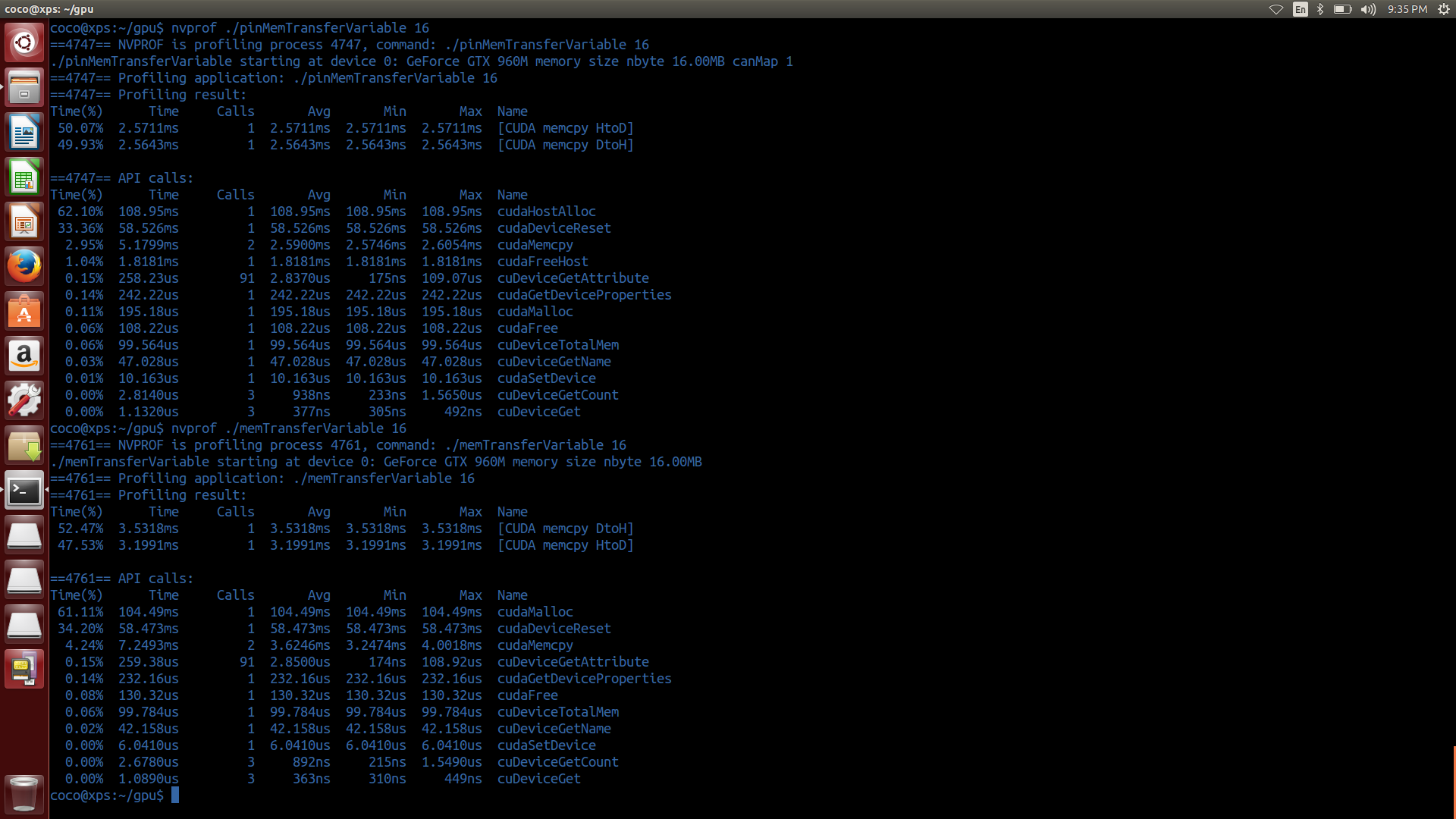
Memory size 4M



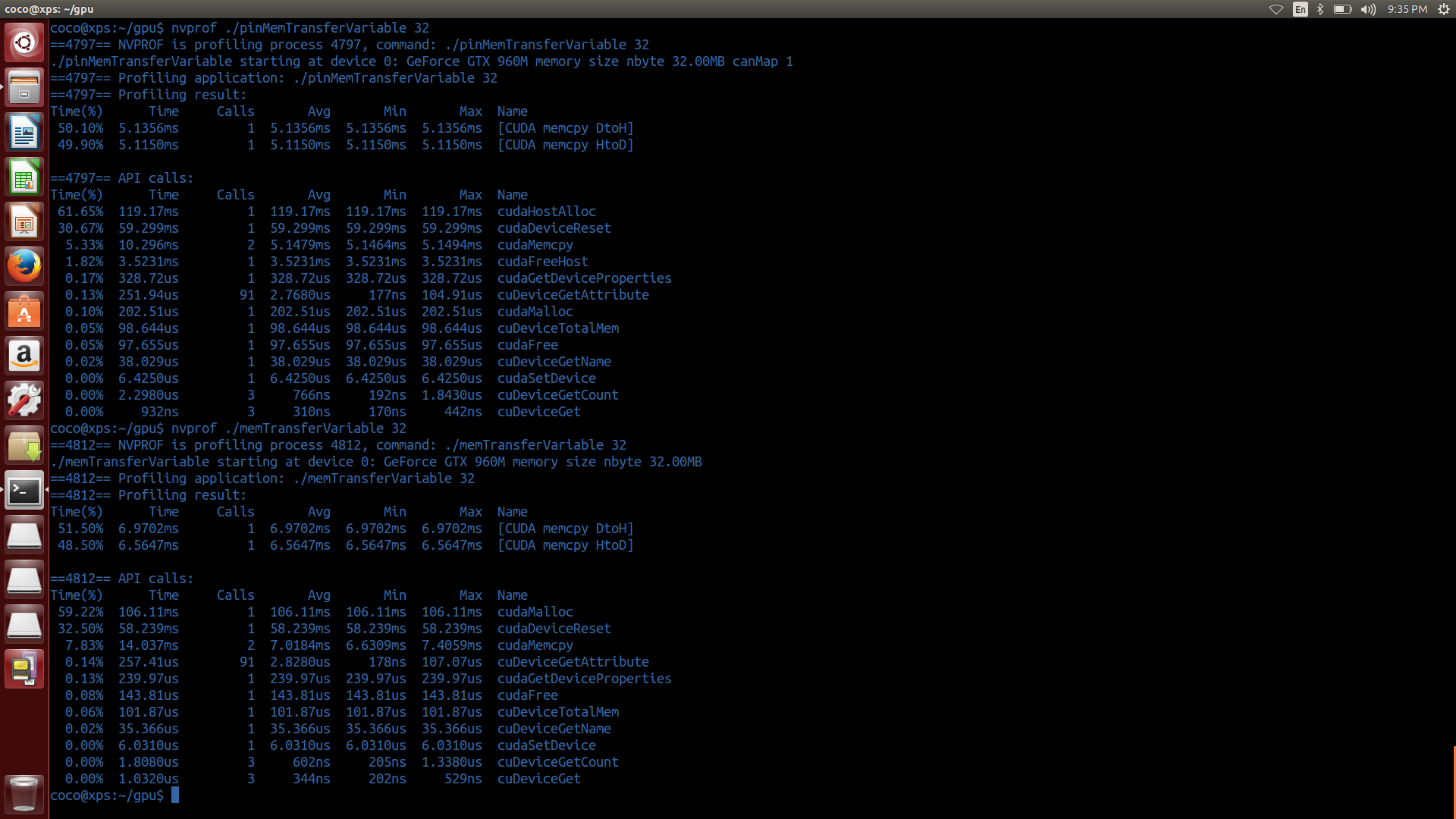
Memory size 8M



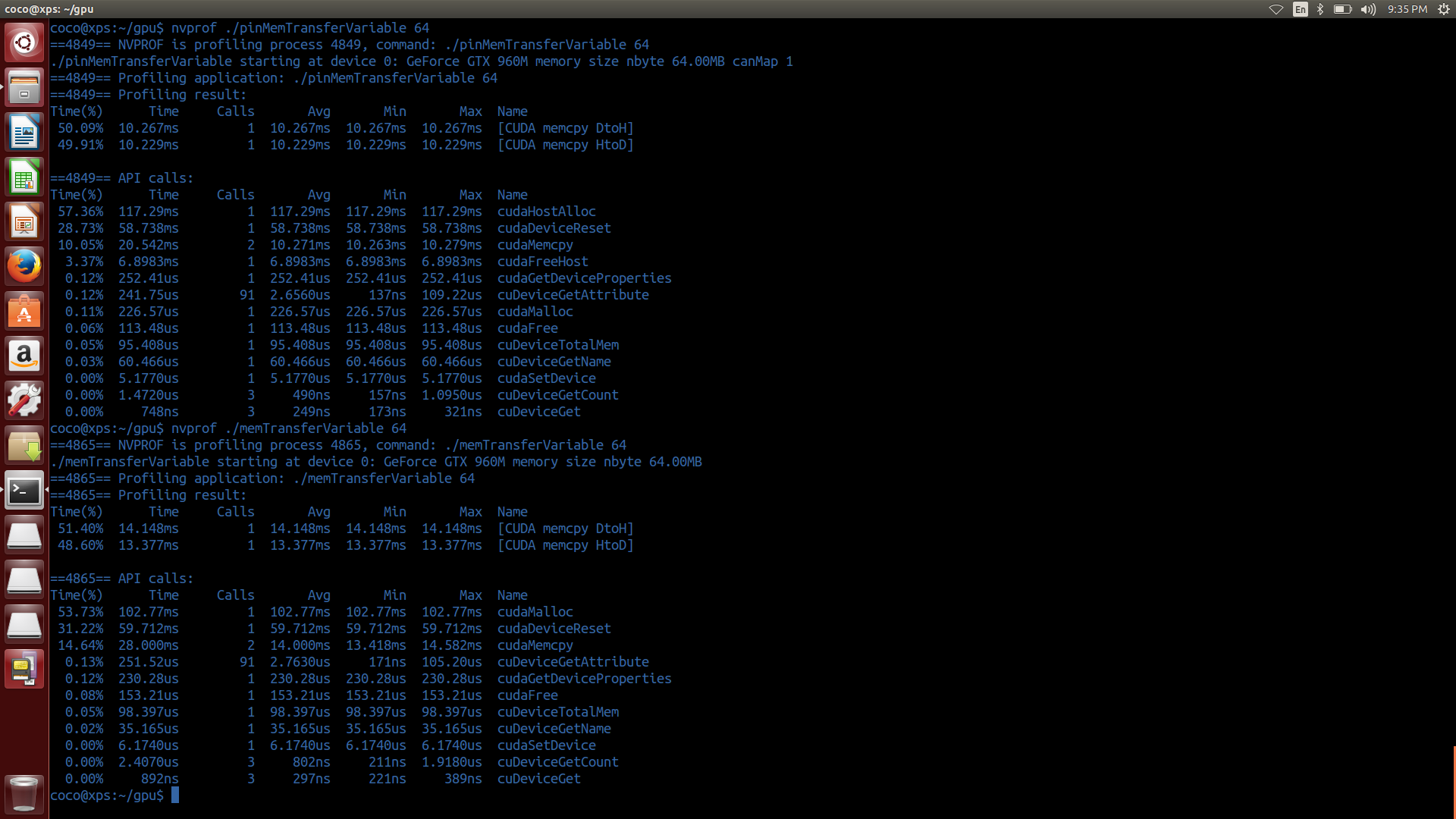
Memory size 16M



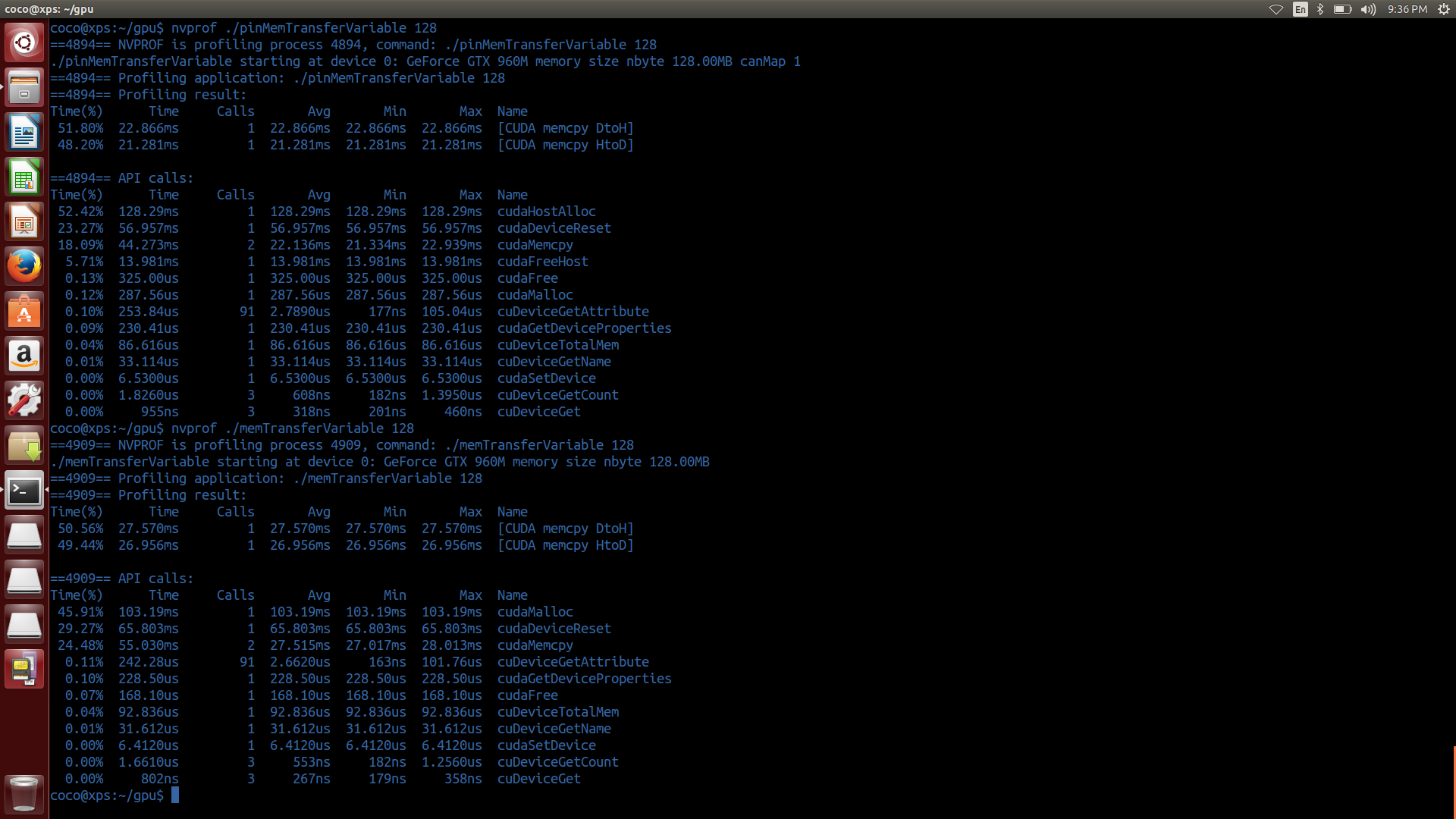
Memory size 32M



Memory size 64M



Memory size 128M



3. At the file globalVariable.cu, replace the following functions:

cudaMemcpyToSymbol()

cudaMemcpyFromSymbol()

with the function

cudaMemcpy() and run the modified program.

You will need to acquire the address of global variable using

cudaGetSymbolAddress() .

cudaMemcpyToSymbol or cudaMemcpyFromSymbol only need the symbol .   
but when you use the cudaMemcpy, you need the address of the device memory. before cudaMemcpy, you should call the cudaGetSymbolAddress() to get the address of the device memory.

[source code]

globalVariable\_2.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* An example of using a statically declared global variable (devData) to store

\* a floating-point value on the device.

\*/

\_\_device\_\_ float devData;

// print the current value of the global variable and increase by 2.0

\_\_global\_\_ void checkGlobalVariable()

{

// display the original value

printf("Device: the value of the global variable is %f\n", devData);

// alter the value

devData += 2.0f;

}

int main(void)

{

// initialize the global variable

float value = 3.14f;

float\* devPtr;

// get the device global variable address

// return : cudaError\_t

// parameter : void\*\* devPtr, const char\* symbol

// return in (\*devPtr) the address of symbol (symbol) on the devce

CHECK(cudaGetSymbolAddress((void\*\*)&devPtr,devData));

// transfer data from the host 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(devPtr, &value, sizeof(float),cudaMemcpyHostToDevice));

printf("Host: copied %f to the global variable\n", value);

// invoke the kernel

checkGlobalVariable<<<1, 1>>>();

// transfer data from the device to the host

// 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(&value, devPtr, sizeof(float),cudaMemcpyDeviceToHost));

printf("Host: the value changed by the kernel to %f\n", value);

// reset device

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

}

4. Analyze and explain the program ‘readSegment.cu and run the program for the following offsets 0,8,16,32,64,128, 160,192, 256. Explain your results.

When you enable L1 cache, the bandwidth from L1 to sm is 128byte. When you access the data that align to your cache line, only one data access is enough, but when you access the data that don’t align to your cache line, additional access need to complete the operation.

As you can see, if the offset is not multiple of 32, you access the cache additionaly

|  |  |  |
| --- | --- | --- |
| offset | warmup | readOffset |
| 0 | 0.000349sec | 0.000205sec |
| 8 | 0.000367sec | 0.000194sec |
| 16 | 0.000368sec | 0.000206sec |
| 32 | 0.000349sec | 0.000207sec |
| 64 | 0.000349sec | 0.000205sec |
| 128 | 0.000360sec | 0.000206sec |
| 160 | 0.000348sec | 0.000192sec |
| 192 | 0.000348sec | 0.000191sec |
| 256 | 0.000350sec | 0.000205sec |

[source code]

readSegment.cu

#include "common.h"

#include <cuda\_runtime.h>

#include <stdio.h>

/\*

\* This example demonstrates the impact of misaligned reads on performance by

\* forcing misaligned reads to occur on a float\*.

\*/

// parameter : float \* hostRef, float\* gpuRef, const int N

// compare the data pointed by (hostRef) and the data pointed by (gpuRef). (N) is the number of compared data

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

{

double epsilon = 1.0E-8;

bool match = 1;

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

{

if (abs(hostRef[i] - gpuRef[i]) > epsilon)

{

match = 0;

printf("different on %dth element: host %f gpu %f\n", i, hostRef[i],

gpuRef[i]);

break;

}

}

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

}

// parameter : float \*ip, int size

// init the data pointed by (ip). (size) is the number of inited data

void initialData(float \*ip, int size)

{

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

{

ip[i] = (float)( rand() & 0xFF ) / 100.0f;

}

return;

}

// parameter : float \*A, float \*B ,float \*C

// skip the (offset), then add the data pointed by (A) to the data pointed by (B) and save that data to the (C) in the host

void sumArraysOnHost(float \*A, float \*B, float \*C, const int n, int offset)

{

for (int idx = offset, k = 0; idx < n; idx++, k++)

{

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

}

}

// parameter : float \* A, float \* B, float \*C

// skip the (offset), then add the data pointed by (A) to the data pointed by (B) and save that data to the (C) in the device

\_\_global\_\_ void warmup(float \*A, float \*B, float \*C, const int n, int offset)

{

unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned int k = i + offset;

if (k < n) C[i] = A[k] + B[k];

}

// parameter : float \* A, float \* B, float \*C

// skip the (offset), then add the data pointed by (A) to the data pointed by (B) and save that data to the (C) in the device

\_\_global\_\_ void readOffset(float \*A, float \*B, float \*C, const int n,

int offset)

{

unsigned int i = blockIdx.x \* blockDim.x + threadIdx.x;

unsigned int k = i + offset;

if (k < n) C[i] = A[k] + B[k];

}

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

{

// set up device

int dev = 0;

cudaDeviceProp deviceProp;

// 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));

// get device information

// return : cudaError\_t

// parameter : cudaDeviceProp\* prop , int device

// returns cudaGetDeviceProperties of device

CHECK(cudaGetDeviceProperties(&deviceProp, dev));

printf("%s starting reduction at ", argv[0]);

printf("device %d: %s ", dev, deviceProp.name);

// set up array size

int nElem = 1 << 20; // total number of elements to reduce

printf(" with array size %d\n", nElem);

size\_t nBytes = nElem \* sizeof(float);

// set up offset for summary

int blocksize = 512;

int offset = 0;

// get the offset from the argument

if (argc > 1) offset = atoi(argv[1]);

// get the blocksize from the argument

if (argc > 2) blocksize = atoi(argv[2]);

// execution configuration

dim3 block (blocksize, 1);

dim3 grid ((nElem + block.x - 1) / block.x, 1);

// allocate host memory

float \*h\_A = (float \*)malloc(nBytes);

float \*h\_B = (float \*)malloc(nBytes);

float \*hostRef = (float \*)malloc(nBytes);

float \*gpuRef = (float \*)malloc(nBytes);

// initialize host array

initialData(h\_A, nElem);

memcpy(h\_B, h\_A, nBytes);

// summary at host side

sumArraysOnHost(h\_A, h\_B, hostRef, nElem, offset);

// allocate device memory

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

// allocate the device memory

// 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((float\*\*)&d\_A, nBytes));

CHECK(cudaMalloc((float\*\*)&d\_B, nBytes));

CHECK(cudaMalloc((float\*\*)&d\_C, nBytes));

// copy data from host to 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(d\_A, h\_A, nBytes, cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(d\_B, h\_A, nBytes, cudaMemcpyHostToDevice));

// kernel 1:

double iStart = seconds();

warmup<<<grid, block>>>(d\_A, d\_B, d\_C, nElem, offset);

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceSynchronize());

double iElaps = seconds() - iStart;

printf("warmup <<< %4d, %4d >>> offset %4d elapsed %f sec\n", grid.x,

block.x, offset, iElaps);

// return : cudaError\_t

// parameter : void

// return the last error

CHECK(cudaGetLastError());

iStart = seconds();

readOffset<<<grid, block>>>(d\_A, d\_B, d\_C, nElem, offset);

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceSynchronize());

iElaps = seconds() - iStart;

printf("readOffset <<< %4d, %4d >>> offset %4d elapsed %f sec\n", grid.x,

block.x, offset, iElaps);

// return : cudaError\_t

// parameter : void

// return the last error

CHECK(cudaGetLastError());

// copy kernel result back to host side and check device results

// 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(gpuRef, d\_C, nBytes, cudaMemcpyDeviceToHost));

checkResult(hostRef, gpuRef, nElem - offset);

// free host and device 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(h\_A);

free(h\_B);

// reset device

// return : cudaError\_t

// parameter : void

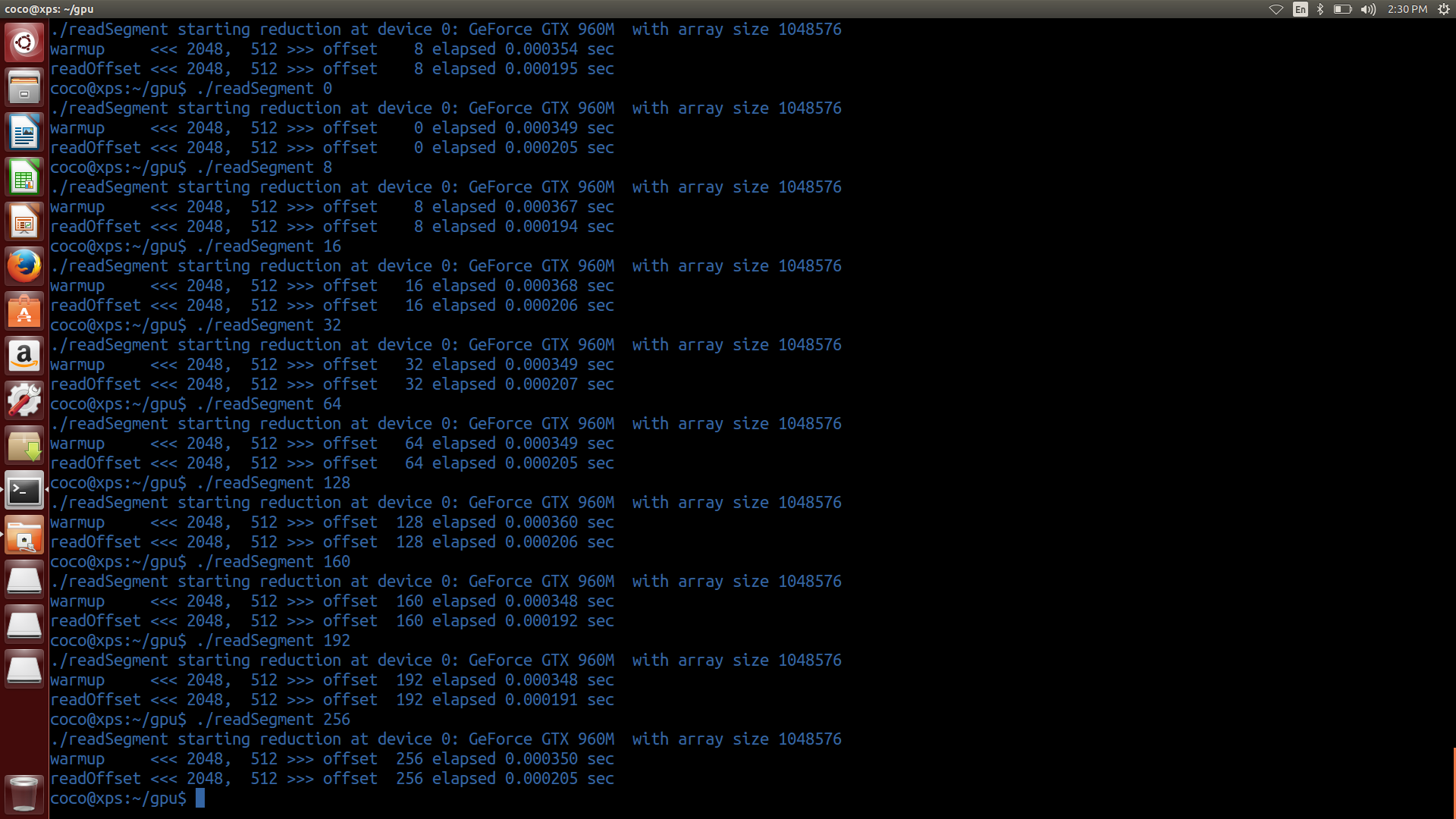
// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return EXIT\_SUCCESS;

}

[screen shot]



5. Run the following command:

$nvprof --metrics gld\_efficiency –metrics gld\_throughput ./readSegment 0 , ./readSegment 11, ./readSegment 128.

Explain your results.

gld\_efficiency : ratio of requested global memory load throughput to required global memory load throughput

gld\_throughput : global memory load throughput

when you access the aligned data, the load efficiency is 100% cause there is no need to additional access to the cache. But when you access the non-aligned data, the load efficiency is not 100% cause there is additional access to the cache and it cause more throughput (maybe there is pre-fetching unit to hide the memory latency, so it increase the throughput).

[screen shot]

