**Problem 1.**

**When I allocate the device memory using cudaMalloc with nx = 15,000 & ny = 15,000. It tell me the error that you don’t have enough memory. So, to check the available memory, I use the cudaGetDeviceProperties API. And I knew that my gpu has 1.96 Gbytes (that is not enough to run the current program). So I reduce the problem size nx =512 & ny = 512**

**[source code]**

#include <stdio.h>

#include "common.h"

// matrix multiplcation kernel function

// 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 size)

\_\_global\_\_ void matrixMul (int \*A, int \*B, int \*C, int nx)

{

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

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

// get the current index in matrix C

unsigned int destIdx = iy \* nx + ix;

if (ix < nx && iy < nx)

{

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

{

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

}

}

}

// host program to check the 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 size)

// return : int (1 : correct 0: not correct)

int check (int\* A, int\* B, int\* C, int \* hostCheck,int size)

{

// calculate the answer

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++)

{

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

}

}

}

// compare the answer

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

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

{

if (hostCheck[col\*size + row] != C[col\*size + row])

{

printf("[%d , %d] host : %d, device : %d\n",col,row,hostCheck[col\*size + row],C[col\*size + row]);

return 0;

}

}

return 1;

}

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

{

int\* A;

int\* B;

int\* C;

int\* hostCheck;

int nx = 512;

int ny = 512;

int dimx = 32;

int dimy = 32;

int size = nx \* ny;

const int BufferSize = size \* sizeof(int);

dim3 block(dimx,dimy);

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

printf("grid [%d %d], block [%d %d] \n",grid.x,grid.y,block.x,block.y);

A = (int\*)malloc(BufferSize);

B = (int\*)malloc(BufferSize);

C = (int\*)malloc(BufferSize);

hostCheck = (int\*)malloc(BufferSize);

// init the value

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

{

A[i] = i % 1000;

B[i] = i % 1000;

C[i] = 0;

hostCheck[i] = 0;

}

int\* d\_A;

int\* d\_B;

int\* 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((void\*\*)&d\_A,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_B,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_C,BufferSize));

// fills the first (count) bytes of the memory area pointed to by (devPtr) with the constant byte value (value)

// return : cudaError\_t

// parameter : void\* devPtr, int value, size\_t count

CHECK(cudaMemset(d\_C,0,BufferSize));

// 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,A,BufferSize,cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(d\_B,B,BufferSize,cudaMemcpyHostToDevice));

// kernel function : run the matrixmultiplication

matrixMul<<<grid,block>>> (d\_A,d\_B,d\_C,nx);

CHECK(cudaMemcpy (C,d\_C,BufferSize,cudaMemcpyDeviceToHost));

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceSynchronize());

// check the answer

if(!check(A,B,C,hostCheck,nx))

{

printf("not correct answer\n");

exit(1);

}

else

{

printf("correct answer\n");

}

// 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(A);

free(B);

free(C);

free(hostCheck);

// reset device

// return : cudaError\_t

// parameter : void

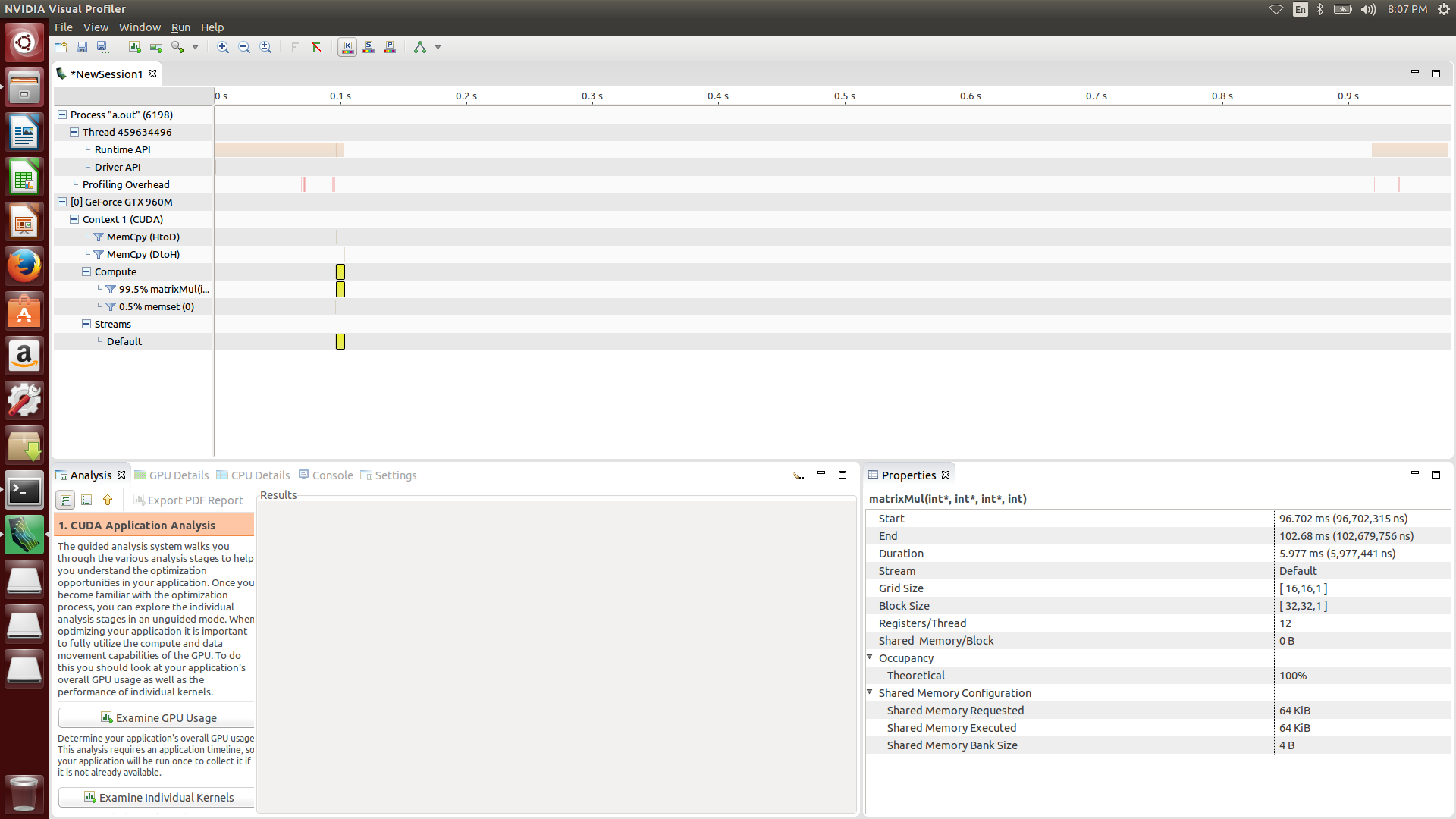
// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return 0;

}

**[screen shot]**

****

**Problem 2.**

**I just change the block size and then calculate the proper grid size. to reduce the problem size, I only choose the square block size.**

**Here is my block configuration and the execution time**

|  |  |
| --- | --- |
| **Block [32,32] Grid [16,16]** | **5.974ms** |
| **Block [16,16] Grid [32,32]** | **5.891ms** |
| **Block [8,8] Grid [64,64]** | **17.675ms** |
| **Block [4,4] Grid [128,128]** | **21.609ms** |
| **Block [2,2] Grid [256,256]** | **58.187ms** |
| **Block [1,1] Grid [512,512]** | **212.151ms** |

**A block consists of warp that have same program flow, and each SM have 1 or 2 block at the same time. So, if the number of block is larger than the available number of block in the device at the same time, some of the blocks have to wait to enter the SM. On the other hand, if you don’t have enough blocks, some of the SM will be idle. So, when you increase total number of block in the device or you don’t have enough blocks, you will see the performance degradation.**

**[source code]**

#include <stdio.h>

#include "common.h"

// matrix multiplcation kernel function

// 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 size)

\_\_global\_\_ void matricMul (int \*A, int \*B, int \*C, int nx)

{

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

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

// get the current index in matrix C

unsigned int destIdx = iy \* nx + ix;

if (ix < nx && iy < nx)

{

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

{

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

}

}

}

// host program to check the 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 size)

// return : int (1 : correct 0: not correct)

int check (int\* A, int\* B, int\* C, int \* hostCheck,int size)

{

// calculate the answer

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++)

{

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

}

}

// compare the answer

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

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

{

if (hostCheck[col\*size + row] != C[col\*size + row])

return 0;

}

return 1;

}

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

{

int\* A;

int\* B;

int\* C;

int\* hostCheck;

int dim = 32;

// change the block configuration

for (dim = 32; dim != 0 ; dim /=2)

{

int nx = 512;

int ny = 512;

int size = nx \* ny;

const int BufferSize = size \* sizeof(int);

dim3 block(dim,dim);

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

A = (int\*)malloc(BufferSize);

B = (int\*)malloc(BufferSize);

C = (int\*)malloc(BufferSize);

hostCheck = (int\*)malloc(BufferSize);

// init the value

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

{

A[i] = i % 1000;

B[i] = i % 1000;

C[i] = 0;

hostCheck[i] = 0;

}

int\* d\_A;

int\* d\_B;

int\* 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((void\*\*)&d\_A,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_B,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_C,BufferSize));

// fills the first (count) bytes of the memory area pointed to by (devPtr) with the constant byte value (value)

// return : cudaError\_t

// parameter : void\* devPtr, int value, size\_t count

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

// 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,A,BufferSize,cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(d\_B,B,BufferSize,cudaMemcpyHostToDevice));

// kernel function : run the matrixmultiplication

matricMul<<<grid,block>>> (d\_A,d\_B,d\_C,nx);

CHECK(cudaMemcpy (C,d\_C,BufferSize,cudaMemcpyDeviceToHost));

// check the answer

if(!check(A,B,C,hostCheck,nx))

{

printf("not correct answer block : [%d,%d] grid : [%d,%d]\n",block.y,block.x,grid.y,grid.x);

exit(1);

}

else

{

printf("correct answer\n");

}

// 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(A);

free(B);

free(C);

free(hostCheck);

// 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());

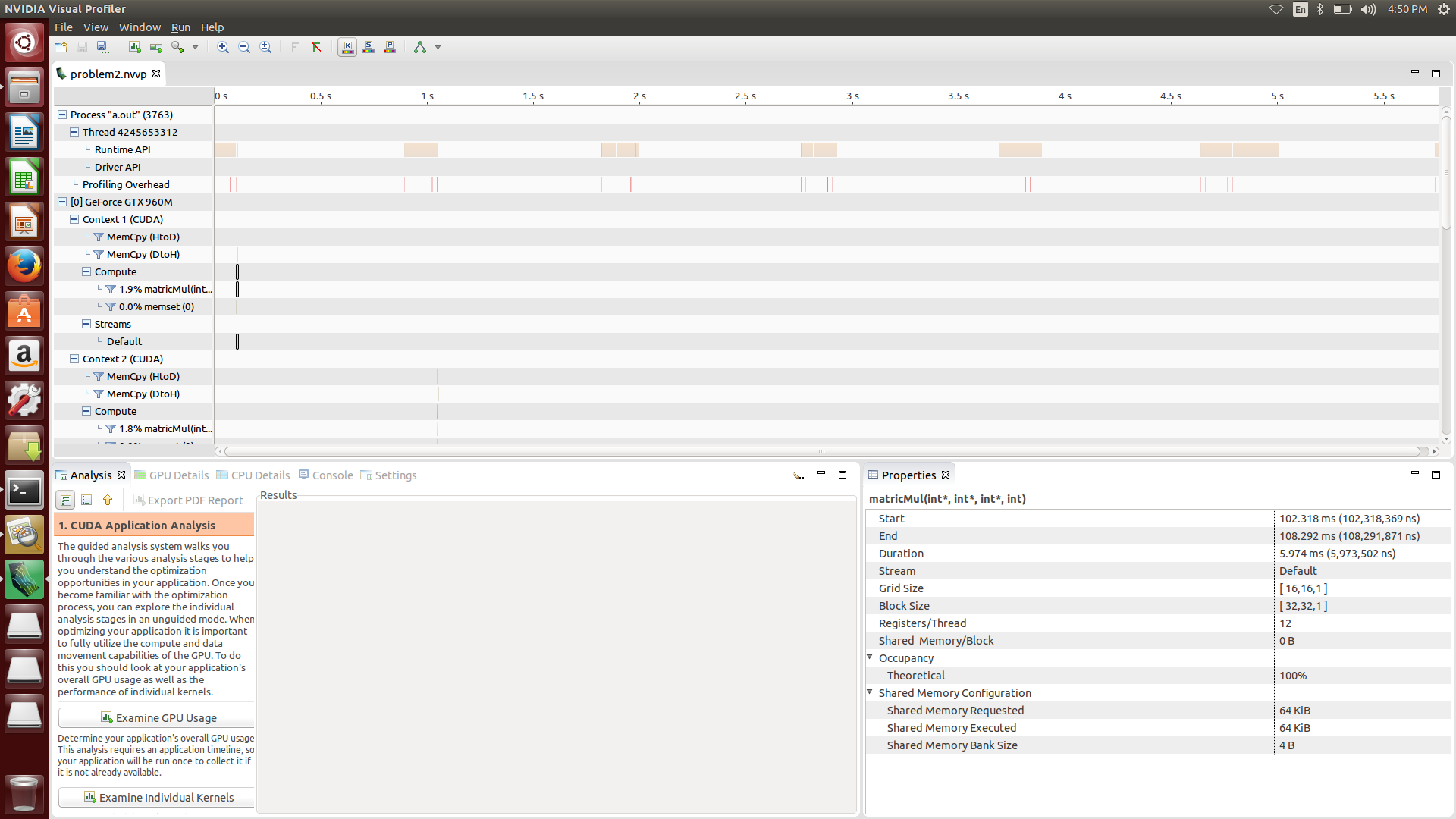
}

return 0;

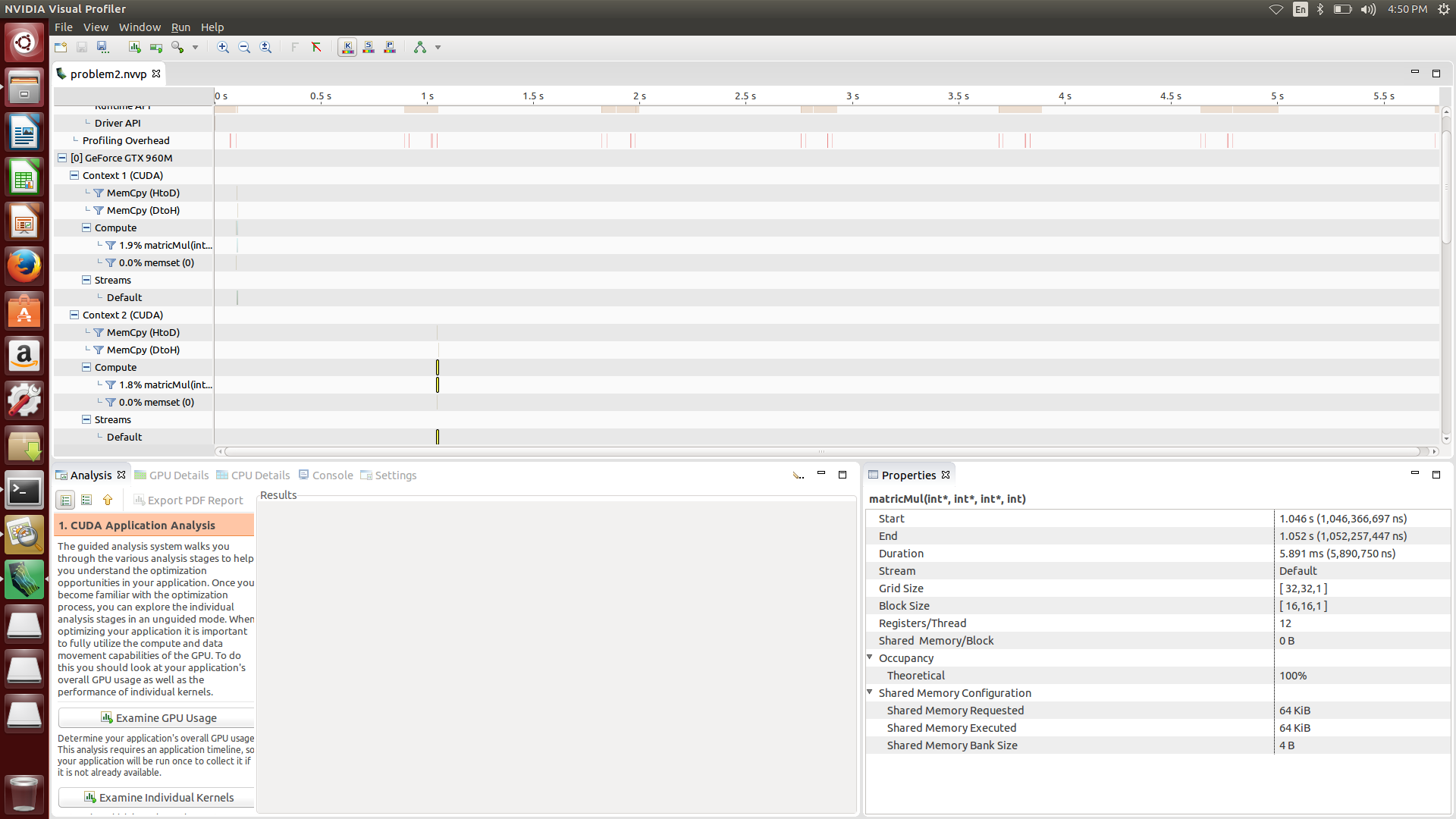
}

**[screen shot]**

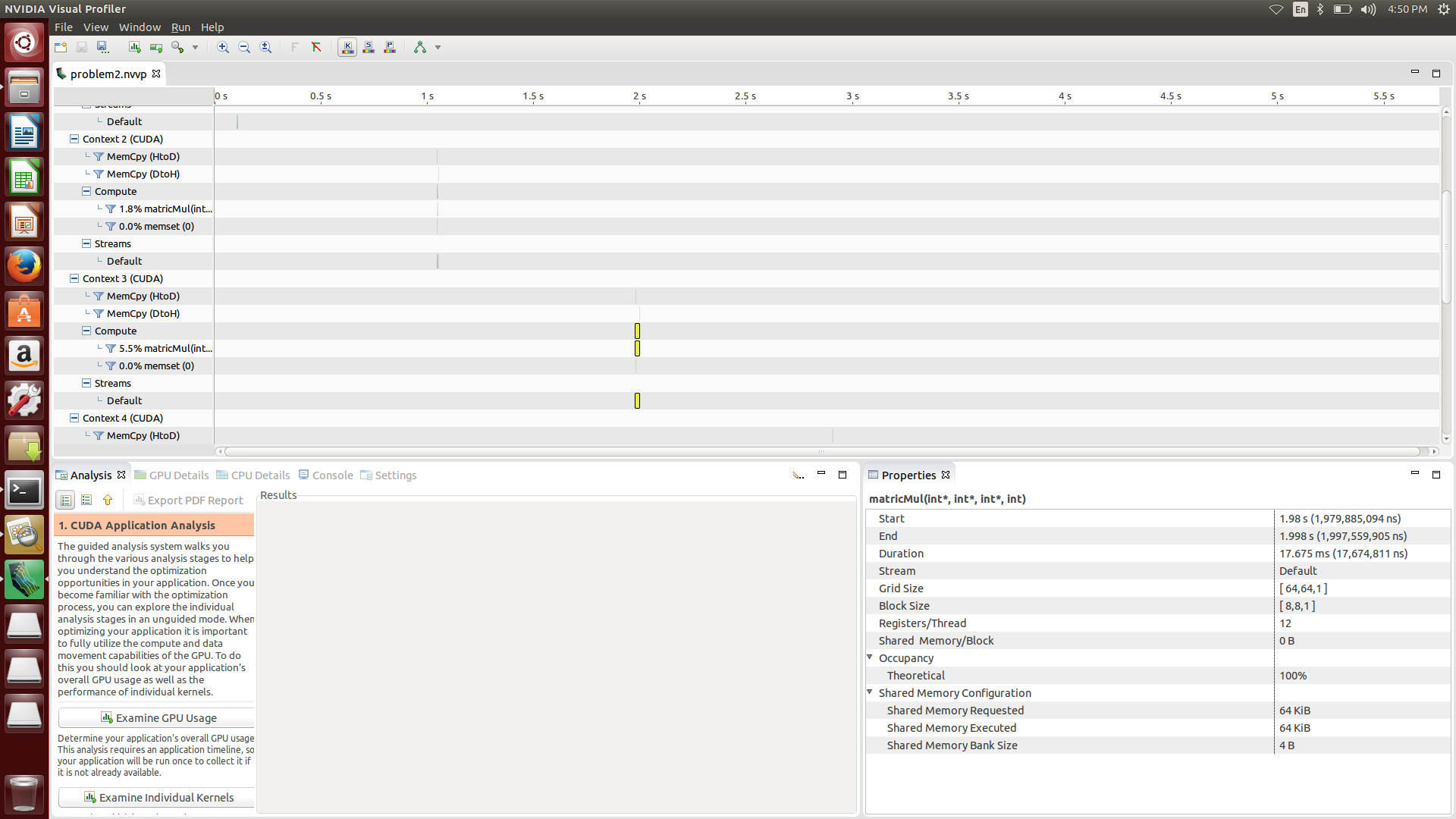
**Block [32,32] Grid [16,16]**

****

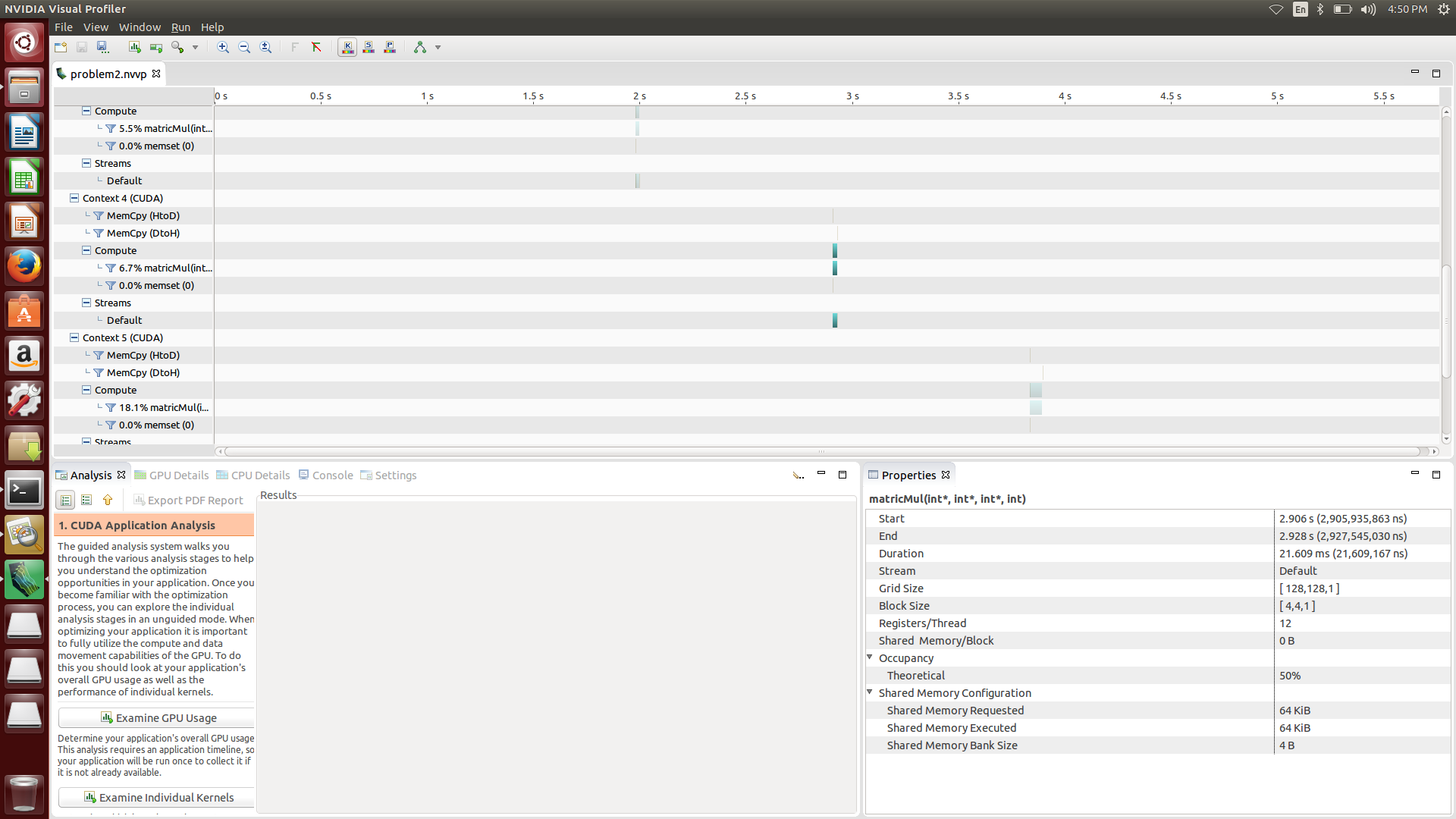
**Block [16,16] Grid [32,32]**

****

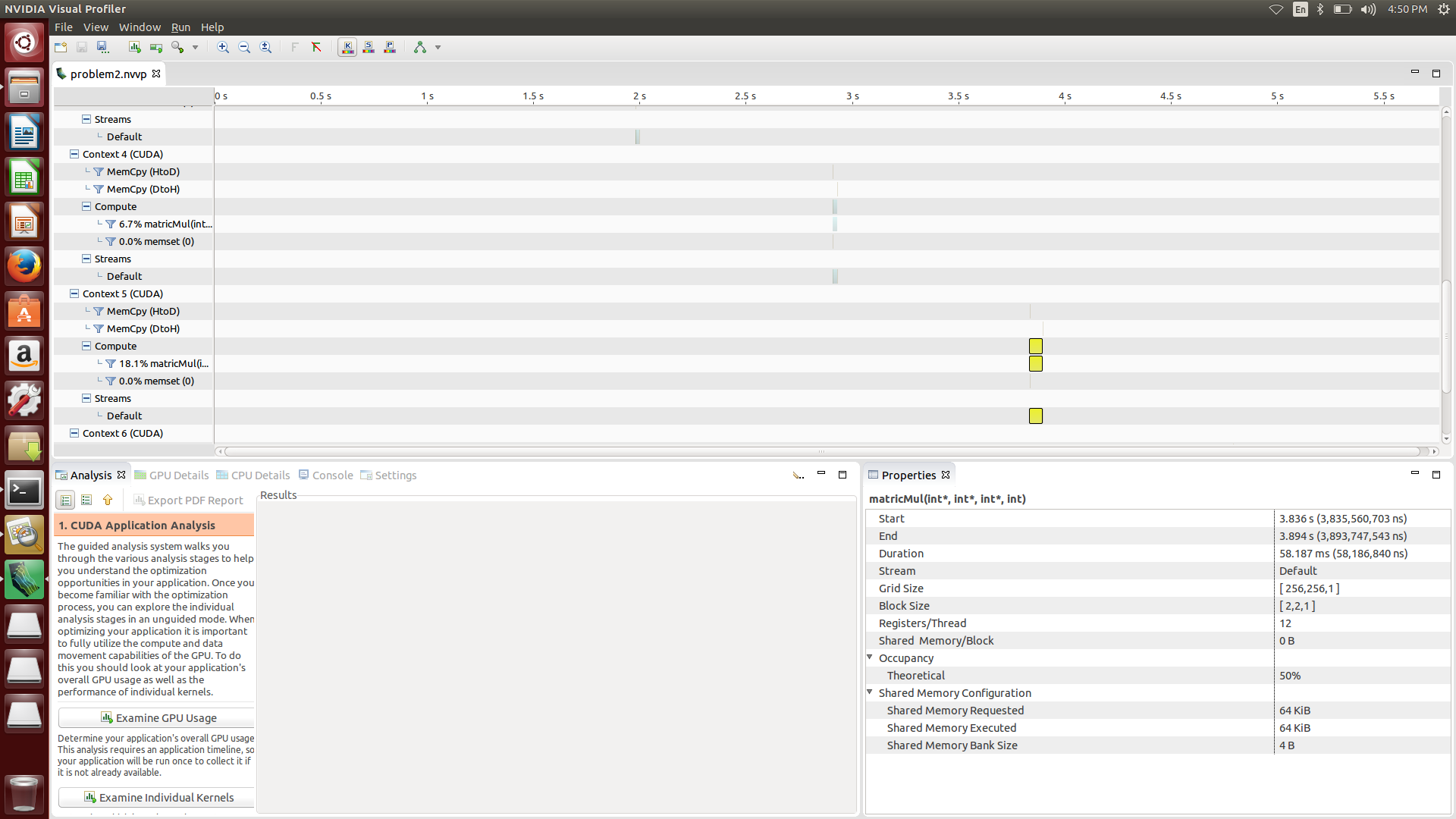
**Block [8,8] Grid [64,64]**

****

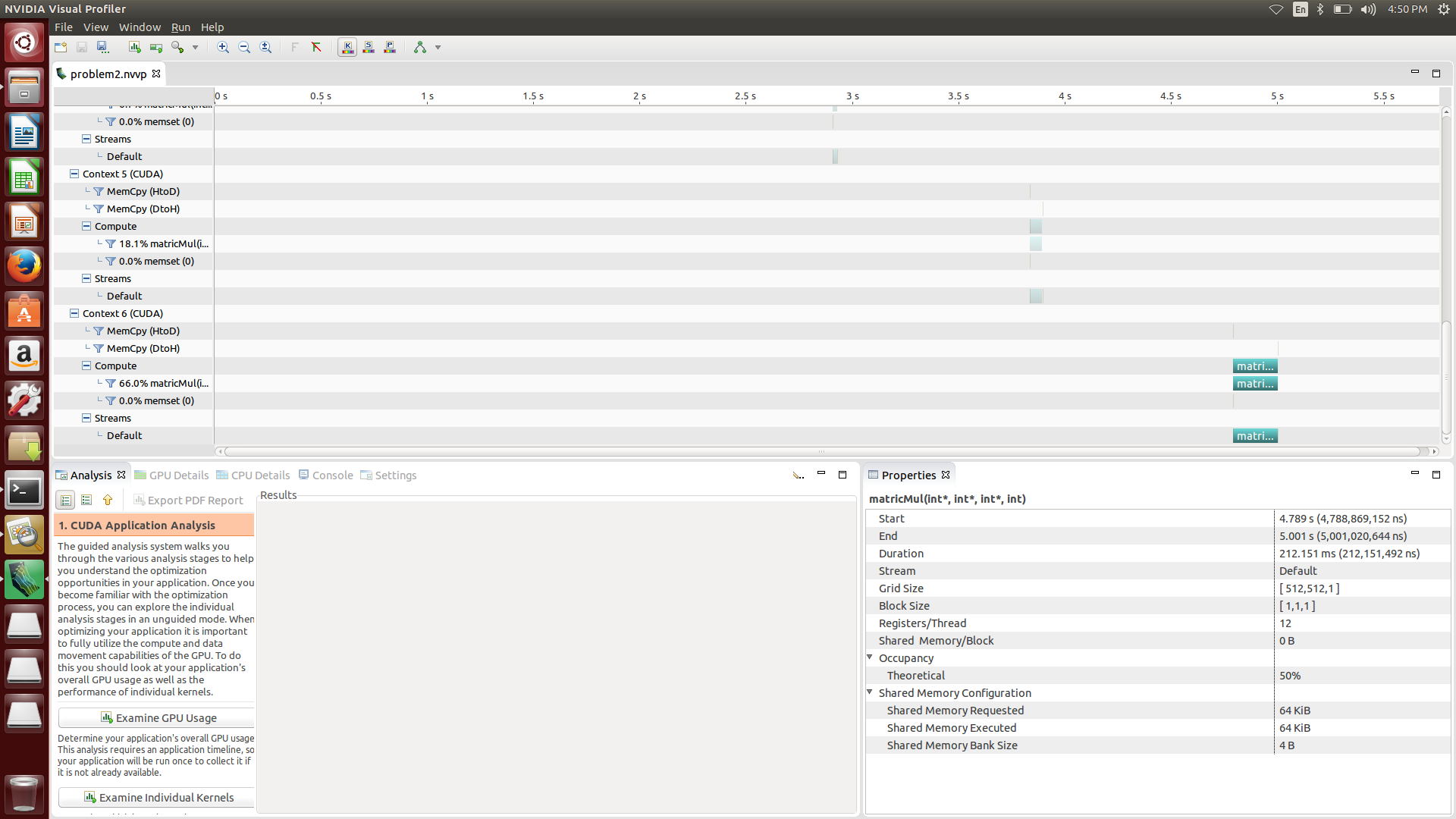
**Block [4,4] Grid [128,128]**

****

**Block [2,2] Grid [256,256]**

****

**Block [1,1] Grid [512,512]**

****

**Problem 3.**

|  |  |  |
| --- | --- | --- |
| **Block [32,32] Grid [16,16]** | **5.974ms** | **1.698ms** |
| **Block [16,16] Grid [32,32]** | **5.891ms** | **1.734ms** |
| **Block [8,8] Grid [64,64]** | **17.675ms** | **2.176ms** |
| **Block [4,4] Grid [128,128]** | **21.609ms** | **7.506ms** |
| **Block [2,2] Grid [256,256]** | **58.187ms** | **49.133ms** |
| **Block [1,1] Grid [512,512]** | **212.151ms** | **337.633ms** |

**As I mentioned in problem 2, when you increase total number of block in the device or you don’t have enough blocks, you will see the performance degradation. This fact also apply to the shared memory.**

**The speed of shared memory is same as the L1 cache. and L1 cache only contain some of the data in global memory. So, When you use shared memory, you can get a better performance than only use in global memory.**

**[source code]**

#include <stdio.h>

#include "common.h"

#define BLOCK\_SIZE 1

// matrix multiplcation kernel function

// 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 size)

\_\_global\_\_ void matrixMul (int\* A, int\* B, int\* C, int size)

{

// shared memory area , we assign the same size of current block size

\_\_shared\_\_ int subA [BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ int subB [BLOCK\_SIZE][BLOCK\_SIZE];

int tx = threadIdx.x;

int ty = threadIdx.y;

// get the current matrix index to be calculated

int idx = blockIdx.x\*BLOCK\_SIZE + tx;

int idy = blockIdx.y\*BLOCK\_SIZE + ty;

int currentVal = 0;

// we calculate the matrix by partitioning the problem into smaller problem. each time we perform the subMatrix multiplication one by one.

// the size of smaller problem is BLOCK\_SIZE

for (int i = 0; i < (size + BLOCK\_SIZE -1) /BLOCK\_SIZE ; i++)

{

// each thread in a block load the data from global memory to the shared memory corresponding their matrix index

if ( idx < size && idy < size)

{

subA[ty][tx] = A[idy\*size + i\*BLOCK\_SIZE + tx];

subB[ty][tx] = B[(i\*BLOCK\_SIZE+ty)\*size + idx];

}

else

{

subA[ty][tx] = 0;

subB[ty][tx] = 0;

}

// wait all of the thread in the block to ensure correct program answer.

\_\_syncthreads();

// calculate subMatrix multiplication

for (int j = 0; j < BLOCK\_SIZE; j++)

currentVal += subA[ty][j] \* subB[j][tx];

// wait all of the thread in the block to ensure correct program answer.

\_\_syncthreads();

}

// finally update the corresponding value to the proper location

if (idx < size && idx < size)

C[idy\*size + idx] = currentVal;

}

// host program to check the 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 size)

// return : int (1 : correct 0: not correct)

int check (int\* A, int\* B, int\* C, int \* hostCheck,int size)

{

// calculate the answer

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++)

{

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

}

}

// compare the answer

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

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

{

if (hostCheck[col\*size + row] != C[col\*size + row])

return 0;

}

return 1;

}

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

int \*A;

int \*B;

int \*C;

int \*hostCheck;

int nx = 512;

int ny = 512;

int size = nx \* ny;

const int BufferSize = size \* sizeof(int);

dim3 block(BLOCK\_SIZE,BLOCK\_SIZE);

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

A = (int\*)malloc(BufferSize);

B = (int\*)malloc(BufferSize);

C = (int\*)malloc(BufferSize);

hostCheck = (int\*)malloc(BufferSize);

// init the value

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

{

A[i] = i % 1000;

B[i] = i % 1000;

C[i] = 0;

hostCheck[i] = 0;

}

int\* d\_A;

int\* d\_B;

int\* 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((void\*\*)&d\_A,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_B,BufferSize));

CHECK(cudaMalloc((void\*\*)&d\_C,BufferSize));

// fills the first (count) bytes of the memory area pointed to by (devPtr) with the constant byte value (value)

// return : cudaError\_t

// parameter : void\* devPtr, int value, size\_t count

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

// 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,A,BufferSize,cudaMemcpyHostToDevice));

CHECK(cudaMemcpy(d\_B,B,BufferSize,cudaMemcpyHostToDevice));

// kernel function : run the matrixmultiplication

matrixMul<<<grid,block>>> (d\_A,d\_B,d\_C,nx);

CHECK(cudaMemcpy (C,d\_C,BufferSize,cudaMemcpyDeviceToHost));

// check the answer

if(!check(A,B,C,hostCheck,nx))

{

printf("not correct answer\n");

exit(1);

}

else

{

printf("correct answer\n");

}

// 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(A);

free(B);

free(C);

free(hostCheck);

// reset device

// return : cudaError\_t

// parameter : void

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

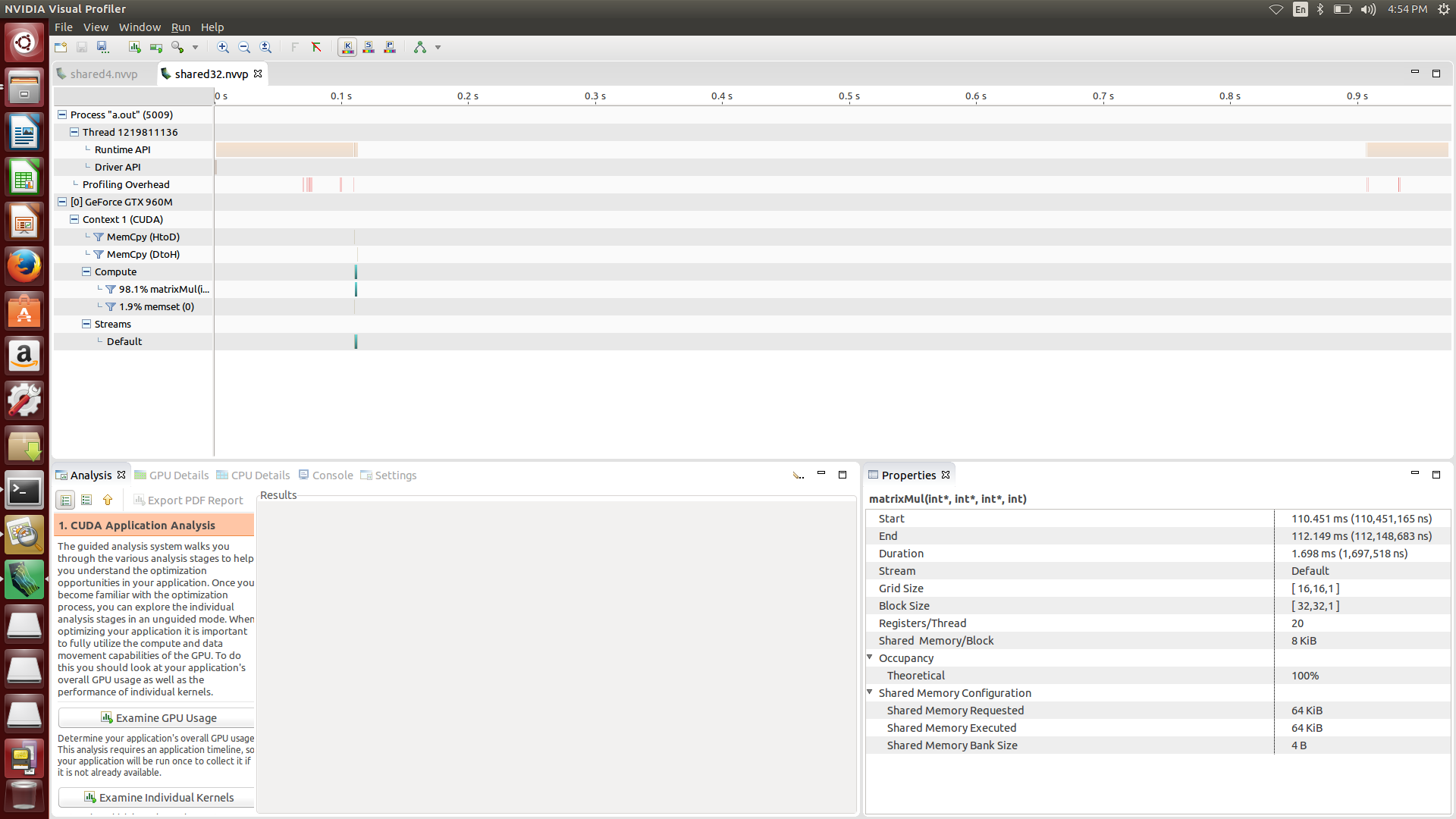
CHECK(cudaDeviceReset());

return 0;

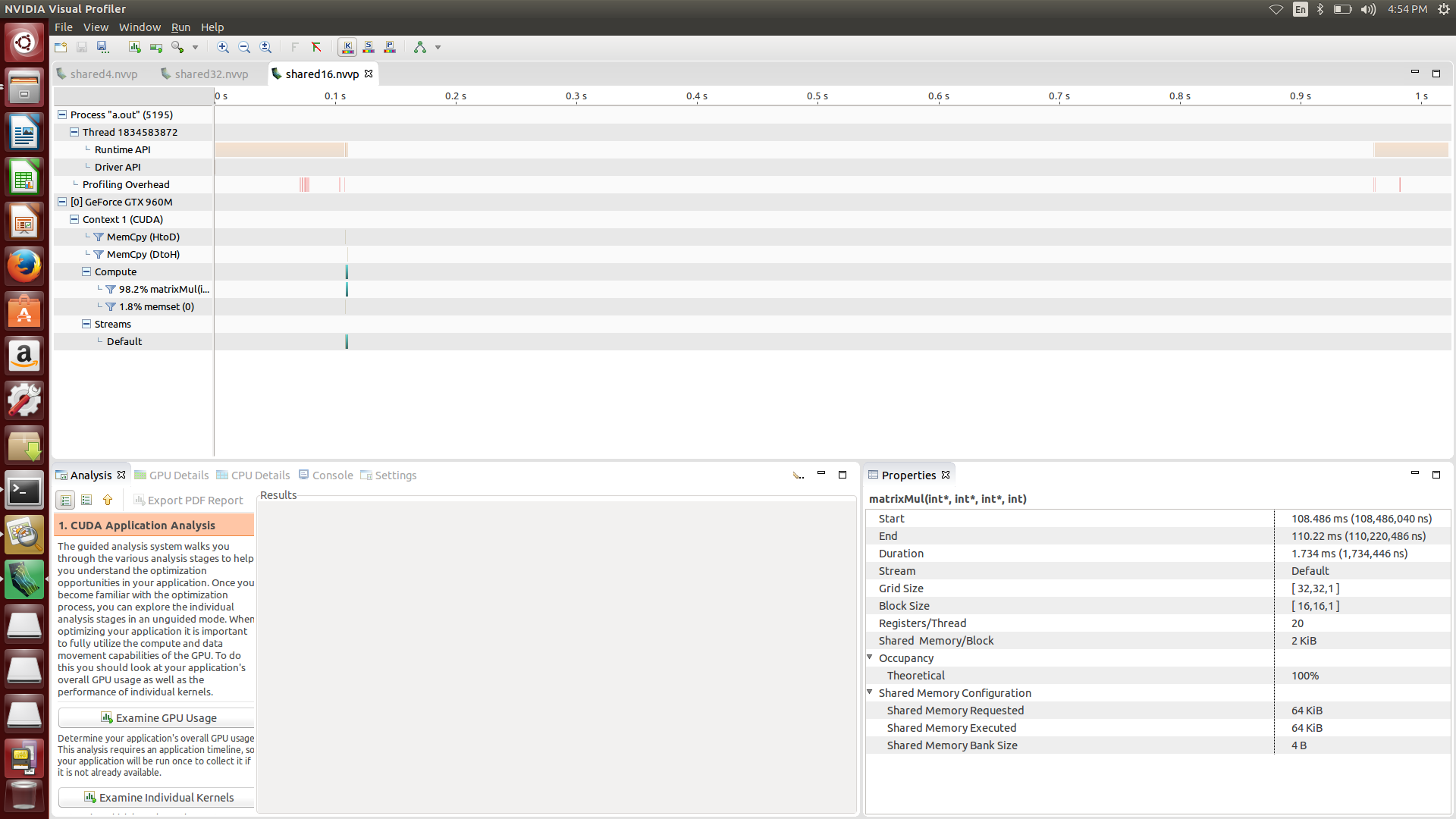
}

**[screen shot]**

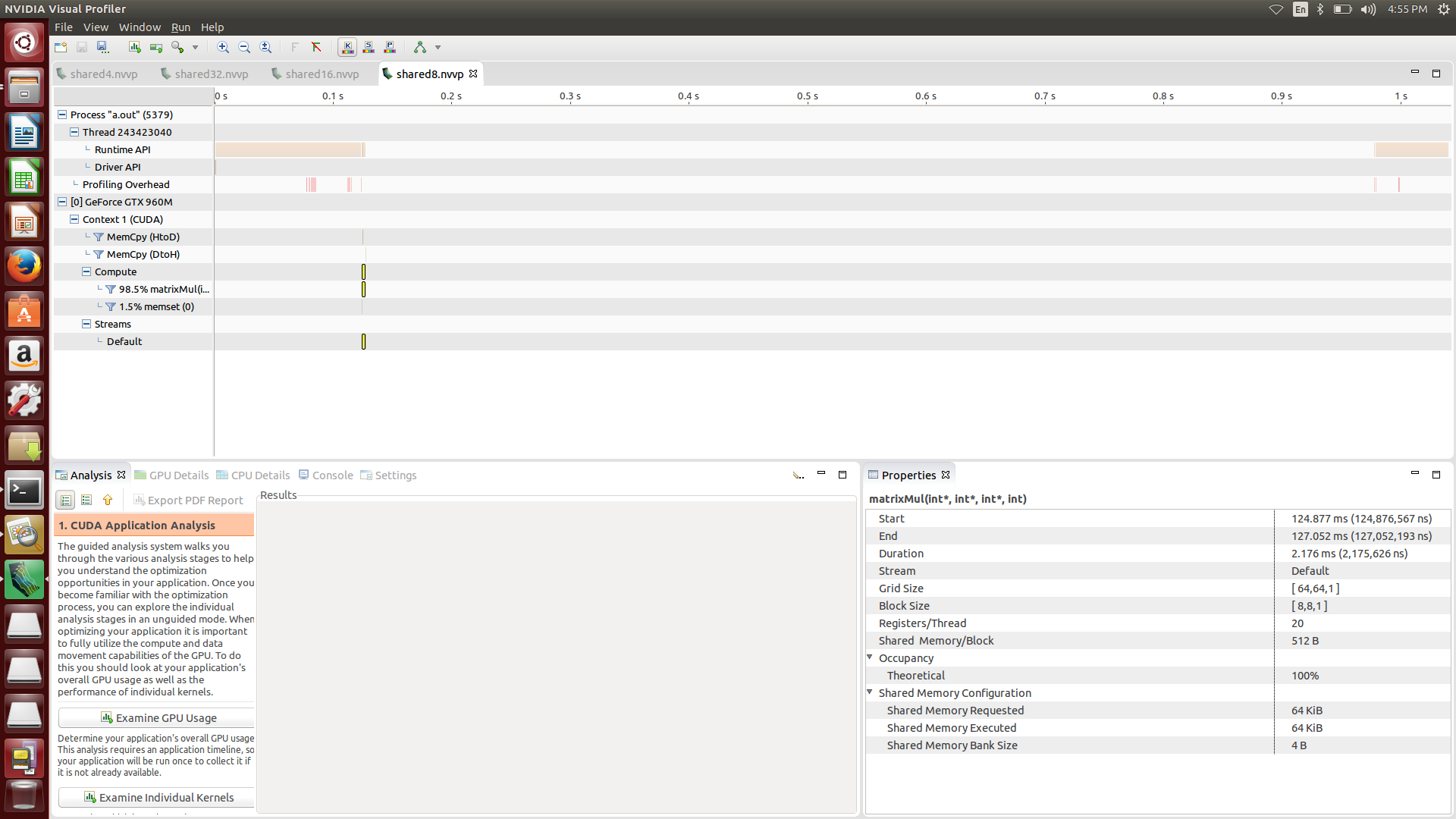
**Block [32,32] Grid [16,16]**

****

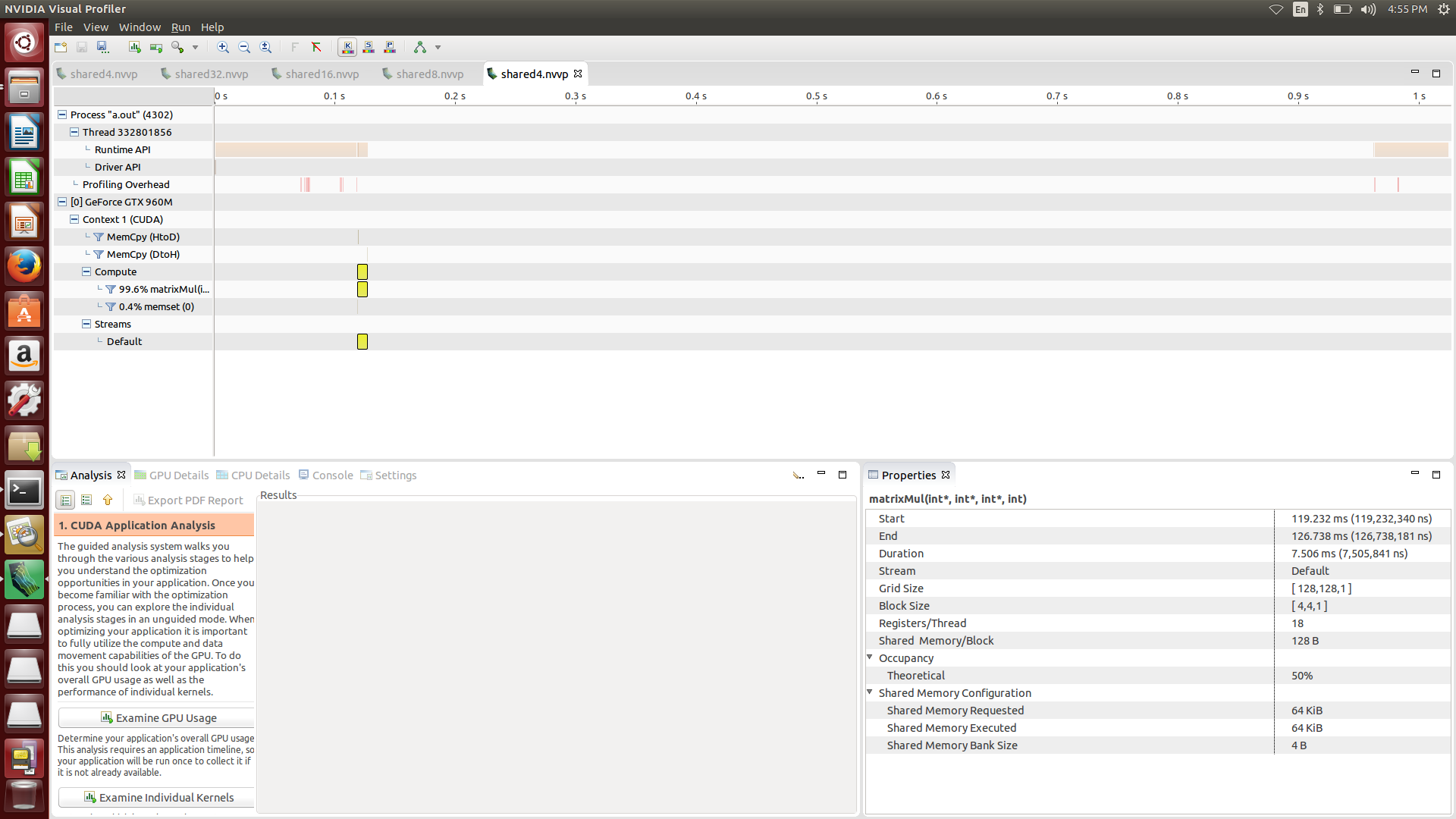
**Block [16,16] Grid [32,32]**

****

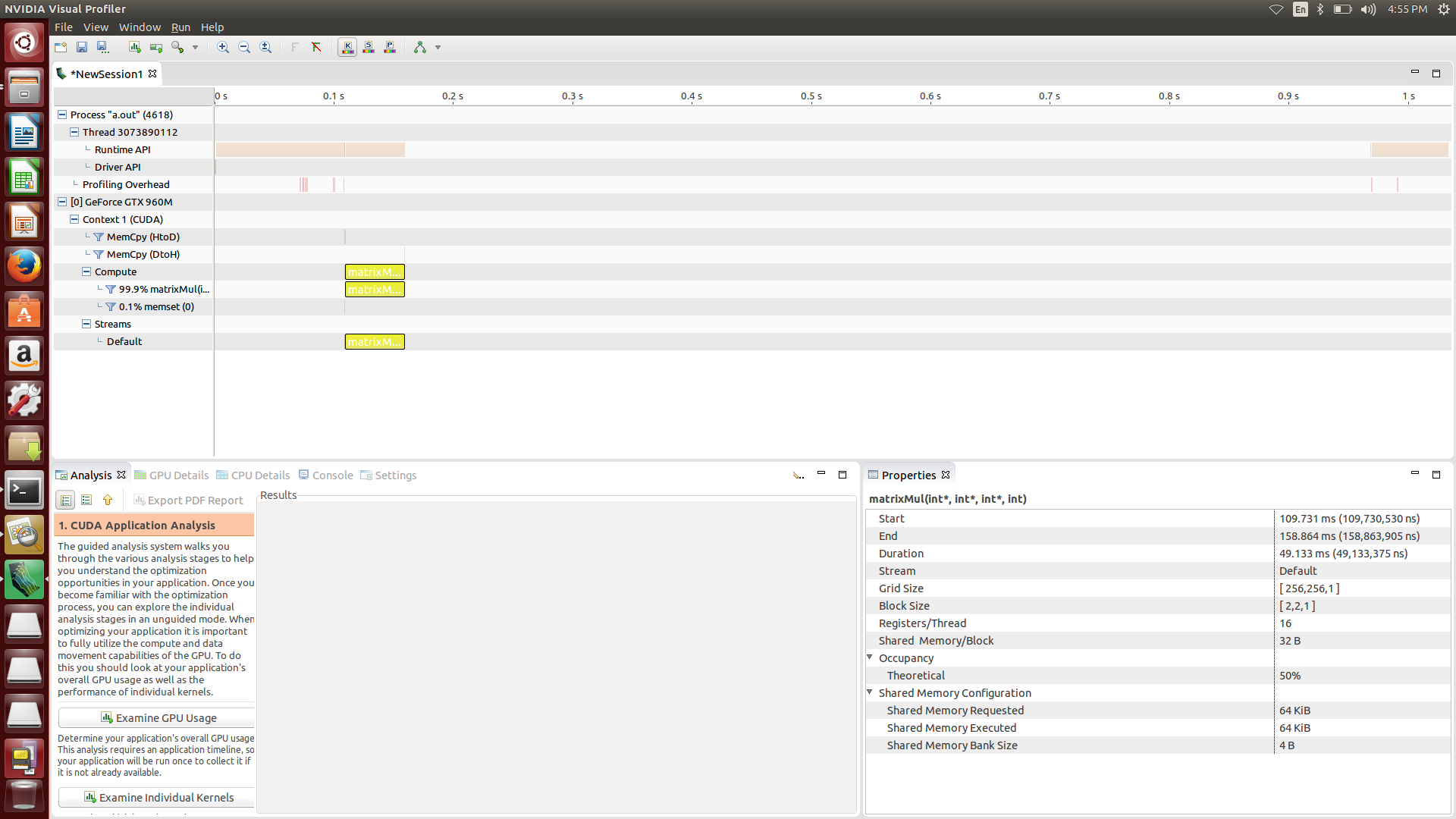
**Block [8,8] Grid [64,64]**

****

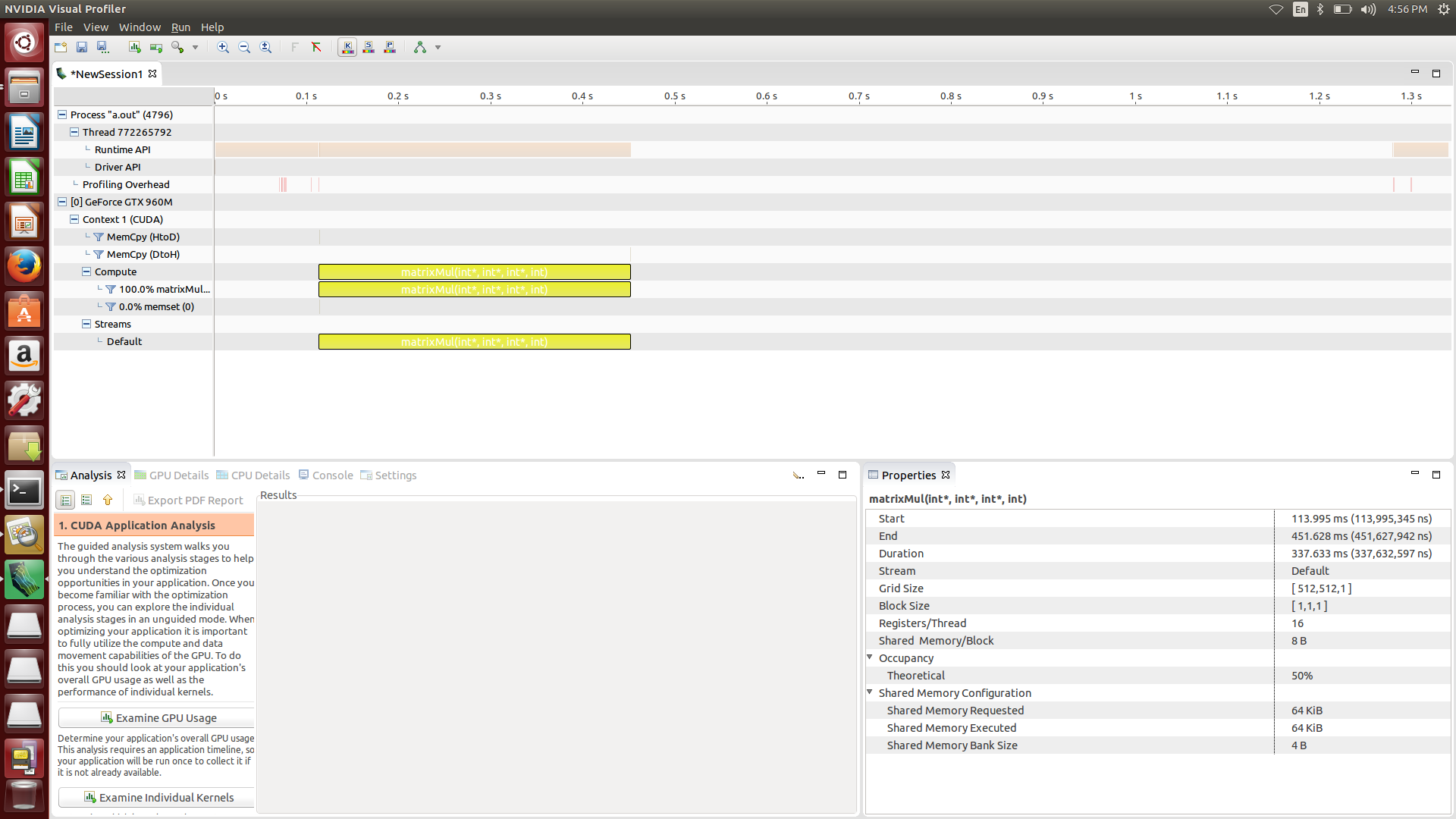
**Block [4,4] Grid [128,128]**

****

**Block [2,2] Grid [256,256]**

****

**Block [1,1] Grid [512,512]**

****

**Problem 4.**

**Unified memory advantage:**

1. **Make cuda Programming easy (make allocate and copy device memory easy)**
2. **To some degree, it is very fast. cause it migrate data on demand between the cpu and gpu.**

**Disadvantage:**

1. **The program with unified memory may be slower than Carefully tuned CUDA program that use overlap execution with data transfers.**

**[source code]**

#include <stdio.h>

#include "common.h"

#define BLOCK\_SIZE 32

// matrix multiplcation kernel function

// 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 size)

\_\_global\_\_ void matrixMul (int\* A, int\* B, int\* C, int size)

{

// shared memory area , we assign the same size of current block size

\_\_shared\_\_ int subA [BLOCK\_SIZE][BLOCK\_SIZE];

\_\_shared\_\_ int subB [BLOCK\_SIZE][BLOCK\_SIZE];

int tx = threadIdx.x;

int ty = threadIdx.y;

// get the current matrix index to be calculated

int idx = blockIdx.x\*BLOCK\_SIZE + tx;

int idy = blockIdx.y\*BLOCK\_SIZE + ty;

int currentVal = 0;

// we calculate the matrix by partitioning the problem into smaller problem. each time we perform the subMatrix multiplication one by one.

// the size of smaller problem is BLOCK\_SIZE

for (int i = 0; i < (size + BLOCK\_SIZE -1) /BLOCK\_SIZE ; i++)

{

// each thread in a block load the data from global memory to the shared memory corresponding their matrix index

if ( idx < size && idy < size)

{

subA[ty][tx] = A[idy\*size + i\*BLOCK\_SIZE + tx];

subB[ty][tx] = B[(i\*BLOCK\_SIZE+ty)\*size + idx];

}

else

{

subA[ty][tx] = 0;

subB[ty][tx] = 0;

}

// wait all of the thread in the block to ensure correct program answer.

\_\_syncthreads();

// calculate subMatrix multiplication

for (int j = 0; j < BLOCK\_SIZE; j++)

currentVal += subA[ty][j] \* subB[j][tx];

// wait all of the thread in the block to ensure correct program answer.

\_\_syncthreads();

}

// finally update the corresponding value to the proper location

if (idx < size && idx < size)

C[idy\*size + idx] = currentVal;

}

// host program to check the 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 size)

// return : int (1 : correct 0: not correct)

int check (int\* A, int\* B, int\* C, int \* hostCheck,int size)

{

// calculate the answer

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++)

{

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

}

}

// compare the answer

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

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

{

if (hostCheck[col\*size + row] != C[col\*size + row])

return 0;

}

return 1;

}

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

int \*A;

int \*B;

int \*C;

int \*hostCheck;

int nx = 512;

int ny = 512;

int size = nx \* ny;

const int BufferSize = size \* sizeof(int);

dim3 block(BLOCK\_SIZE,BLOCK\_SIZE);

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

// allocate unified memory

// return cudaError\_t

// parameter : T\*\* devPtr, size\_t size, unsigned int flags

// allocate memory that will be automatically managed by the unified memory system

CHECK(cudaMallocManaged((void\*\*)&A,BufferSize));

CHECK(cudaMallocManaged((void\*\*)&B,BufferSize));

CHECK(cudaMallocManaged((void\*\*)&C,BufferSize));

CHECK(cudaMallocManaged((void\*\*)&hostCheck,BufferSize));

// init the value

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

{

A[i] = i % 1000;

B[i] = i % 1000;

C[i] = 0;

hostCheck[i] = 0;

}

// kernel function : run the matrixmultiplication

matrixMul<<<grid,block>>> (A,B,C,nx);

// return : cudaError\_t

// parameter : void

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

CHECK(cudaDeviceSynchronize());

// check the answer

if(!check(A,B,C,hostCheck,nx))

{

printf("not correct answer\n");

exit(1);

}

else

{

printf("correct answer\n");

}

// free host and device memory

// return : cudaError\_t

// parameter : void\* devPtr

// free the memory space pointed to by devPtr

CHECK(cudaFree(A));

CHECK(cudaFree(B));

CHECK(cudaFree(C));

CHECK(cudaFree(hostCheck));

// reset device

// return : cudaError\_t

// parameter : void

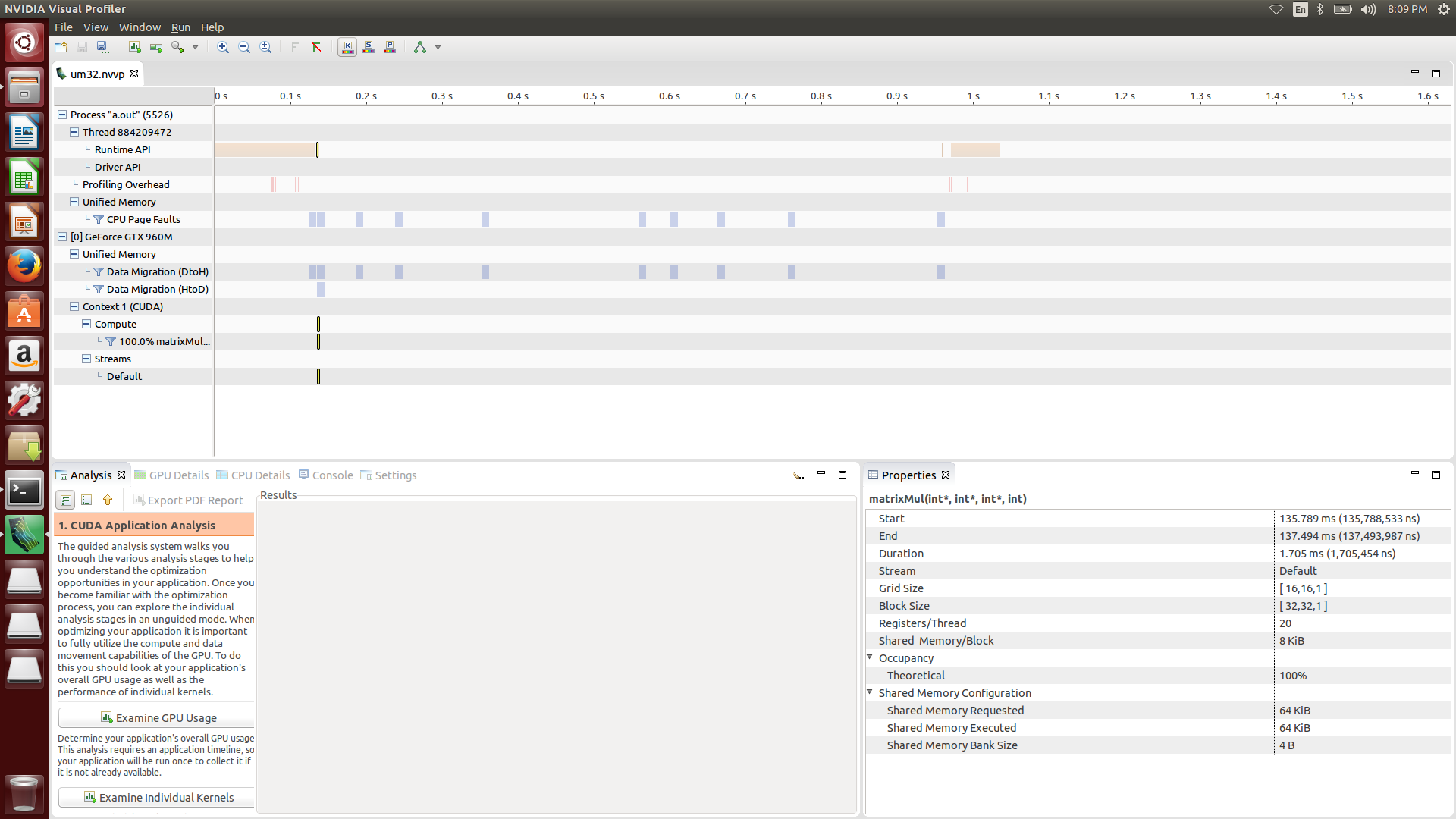
// explicitly destoys and cleans up all resources associated with the current device in the current process

CHECK(cudaDeviceReset());

return 0;

}

**[screen shot]**

****