Profile-Guided Development with OpenACC – C/C++

Carlos Junqueira-Junior
Science Computing Research Engineer

Summary

Summary

Summary

Summary

Summary

Summary

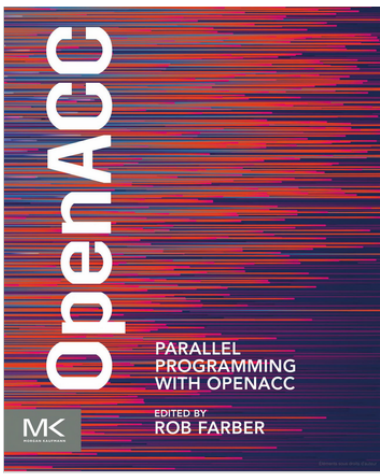
Summary

Summary

Summary

References

Parallel Programming with OpenACC



- Farber, R. (2016).*Parallel programming with OpenACC*. Elsevier.

Introduction

Source files

- The source files can be downloaded at the given GitHub repository:
  - ▸ junqjr's repository

Introduction

Main Goal

- Introduction to OpenACC programming by accelerating a real benchmark application;
- Profiling a code and incrementally improving the application through the addition of OpenACC directives;
- Transform a serial code into a parallel one which can run on both offloaded accelerators, such as Graphical Processing Units (GPU), and multicore Central Processing Units (CPU).

Introduction

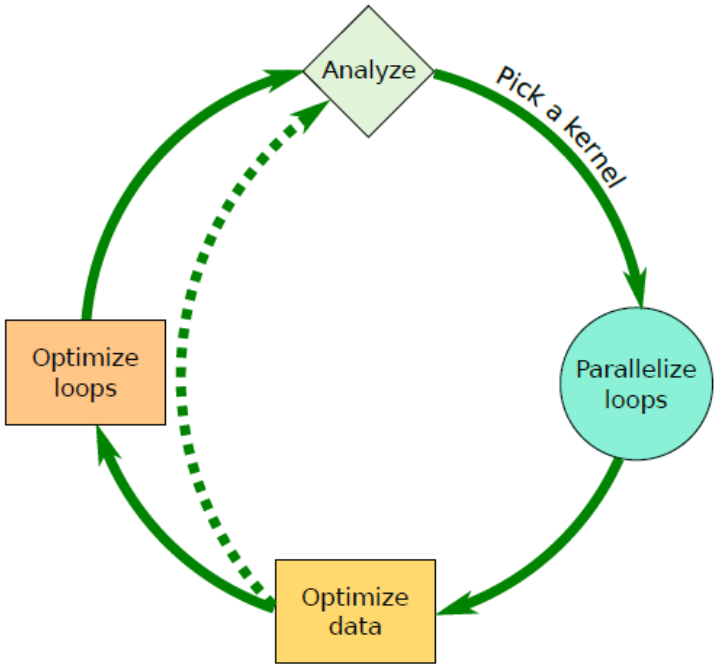
Introduction to Profile-Guided Development with OpenACC

- OpenACC directives provide developers with a means of expressing information on the compiler that is above and beyond what is availble in the standard programming languages;
- OpenACC provides mechanisms for expressing parallelism of loops and the movement of data between distinct physical memories;
- Programmers typically add directives to existing codes incrementally;
- Profile-guided development is a technique that uses performance analysis tools to inform the programmer at each step which parts of the application will deliver the greatest impact once accelerated.
- The PGI compiler along with the PGPROF performance analysis tool is used in the current profile-guided development.

Introduction

Profile-Guided Development Strategy

1. Identify the compute intensive loops;
2. Add OpenACC directives;
3. Optimize data transfers and loops;
4. Repeats 1 to 3 until everything is on the devices.



Introduction

Definitions

- Thread : Execution entity, serial thread of execution that runs any valid C/C++ or Fortran code;
- Worker(OpenACC) : Fine-grain parallelism. Group of threads that can operate together in a SIMD or vector fashion;
- Vector: Group of threads executing the same instruction (SIMT). Vectors cause worker threads to work in lockstep when running vector or SIMD instruction;
- Gang(OpenACC)/Teams(OpenMP) : Coarse-grain parallelism. Groups of workers which operate independently of each other;
- SIMT : Single Instruction Multiple Threads;
- SIMD : Single Instruction Multiple Data;
- Device : Accelerator on which execution can be offloaded (ex: GPU);
- Host : Machine hosting 1 or more accelerators and in charge of execution control;
- Kernel : Piece of code that runs on an accelerator;
- Execution thread : Sequence of kernels to be executed on an accelerator.

Benchmark Code

Conjugate Gradient - C Language

- Iterative method to approximate the solution to a sparse system of linear equations;
- It is not necessary to understand the mathematics of this method to do the exercise;
- The code has two data structures of interest:
  1. Vector structure: A pointer to an array and an integer storing the length of the array;
  2. Matrix structure: Stores a 2-D matrix in compressed sparse row format. Only the non-zero elements of each row are stored, along with metadata representing where the elements would reside in the full matrix.
- The two data structures, along functions which manipulate these data structures, are found in **vector.h** and **matrix.h**.

Benchmark Code

Conjugate Gradient - **main.cpp**

```
20  #include "vector.h"
21  #include "vector_functions.h"
22  #include "matrix.h"
23  #include "matrix_functions.h"
24
25  #define N 150
26  #define MAX_ITERS 100
27  #define TOL 1e-12
28  int main() {
29    vector x,b;
30    vector r,p,Ap;
31    matrix A;
32
33    double one=1.0, zero=0.0;
34    double normr, rtrans, oldtrans, p_ap_dot , alpha, beta;
35    int iter=0;
36
37    //create matrix
38    allocate_3d_poisson_matrix(A,N);
39
40    printf("Rows: %d, nnz: %d\n", A.num_rows, A.row_offsets[A.num_rows]);
41
42    allocate_vector(x,A.num_rows);
43    allocate_vector(Ap,A.num_rows);
44    allocate_vector(r,A.num_rows);
45    allocate_vector(p,A.num_rows);
46    allocate_vector(b,A.num_rows);
47
48    initialize_vector(x,100000);
49    initialize_vector(b,1);
50
51
52    waxpby(one, x, zero, x, p);
53    matvec(A,p,Ap);
54    waxpby(one, b, -one, Ap, r);
55
56    rtrans=dot(r,r);
57    normr=sqrt(rtrans);
```

```
59    double st = omp_get_wtime();
60    do {
61      if(iter==0) {
62        waxpby(one,r,zero,r,p);
63      } else {
64        oldtrans=rtrans;
65        rtrans = dot(r,r);
66        beta = rtrans/oldtrans;
67        waxpby(one,r,beta,p,p);
68      }
69
70      normr=sqrt(rtrans);
71
72      matvec(A,p,Ap);
73      p_ap_dot = dot(Ap,p);
74
75      alpha = rtrans/p_ap_dot;
76
77      waxpby(one,x,alpha,p,x);
78      waxpby(one,r,-alpha,Ap,r);
79
80      if(iter%10==0)
81        printf("Iteration: %d, Tolerance: %.4e\n", iter, normr);
82      iter++;
83    } while(iter<MAX_ITERS && normr>TOL);
84    double et = omp_get_wtime();
85
86    printf("Total Iterations: %d Total Time: %lfs\n", iter, (et-st));
87    printf("Final Tolerance: %.4e\n", normr);
88
89    free_vector(x);
90    free_vector(r);
91    free_vector(p);
92    free_vector(Ap);
93    free_matrix(A);
94
95    return 0;
96  }
```

Benchmark Code

Conjugate Gradient - **vector.h** and **vector_functions.h**

vector.h

```
16 #pragma once
17 #include<cmath>
18
19 struct vector {
20   unsigned int n;
21   double *coefs;
22 };
23
24 void allocate_vector(vector &v, unsigned int n) {
25   v.n=n;
26   v.coefs=(double*)malloc(n*sizeof(double));
27 }
28
29 void free_vector(vector &v) {
30   free(v.coefs);
31   v.n=0;
32 }
33
34 void initialize_vector(vector &v,double val) {
35
36   for(int i=0;i<v.n;i++)
37     v.coefs[i]=val;
38 }
```

vector_functions.h

```
16 #pragma once
17 #include<cstdlib>
18 #include "vector.h"
19
20
21 double dot(const vector& x, const vector& y) {
22   double sum=0;
23   unsigned int n=x.n;
24   double *xcoefs=x.coefs;
25   double *ycoefs=y.coefs;
26
27   for(int i=0;i<n;i++) {
28     sum+=xcoefs[i]*ycoefs[i];
29   }
30   return sum;
31 }
32
33 void waxpby(double alpha, const vector &x, double beta, const vector &y, const vector& w) {
34   unsigned int n=x.n;
35   double *xcoefs=x.coefs;
36   double *ycoefs=y.coefs;
37   double *wcoefs=w.coefs;
38
39   for(int i=0;i<n;i++) {
40     wcoefs[i]=alpha*xcoefs[i]+beta*ycoefs[i];
41   }
42 }
```

Benchmark Code

Conjugate Gradient - **matrix.h**

```
16  #pragma once
17
18  #include<cstdlib>
19
20  struct matrix {
21    unsigned int num_rows;
22    unsigned int nnz;
23    unsigned int *row_offsets;
24    unsigned int *cols;
25    double *coefs;
26  };
27
28
29  void allocate_3d_poisson_matrix(matrix &A, int N) {
30    int num_rows=(N+1)*(N+1)*(N+1);
31    int nnz=27*num_rows;
32    A.num_rows=num_rows;
33    A.row_offsets=(unsigned int*)malloc((num_rows+1)*sizeof(unsigned int));
34    A.cols=(unsigned int*)malloc(nnz*sizeof(unsigned int));
35    A.coefs=(double*)malloc(nnz*sizeof(double));
36
37    int offsets[27];
38    double coefs[27];
39    int zstride=N*N;
40    int ystride=N;
41
42    int i=0;
43    for(int z=-1;z<=1;z++) {
44      for(int y=-1;y<=1;y++) {
45        for(int x=-1;x<=1;x++) {
46          offsets[i]=zstride*z+ystride*y+x;
47          if(x==0 && y==0 && z==0)
48            coefs[i]=27;
49          else
50            coefs[i]=-1;
51          i++;
52        }
53      }
54    }
```

```
56    nnz=0;
57    for(int i=0;i<num_rows;i++) {
58      A.row_offsets[i]=nnz;
59      for(int j=0;j<27;j++) {
60        int n=i+offsets[j];
61        if(n>=0 && n<num_rows) {
62          A.cols[nnz]=n;
63          A.coefs[nnz]=coefs[j];
64          nnz++;
65        }
66      }
67    }
68
69    A.row_offsets[num_rows]=nnz;
70    A.nnz=nnz;
71  }
72
73  void free_matrix(matrix &A) {
74    unsigned int *row_offsets=A.row_offsets;
75    unsigned int * cols=A.cols;
76    double * coefs=A.coefs;
77
78    free(row_offsets);
79    free(cols);
80    free(coefs);
81  }
```

Benchmark Code

Conjugate Gradient - **matrix_functions.h**

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29    for(int i=0;i<num_rows;i++) {
30      double sum=0;
31      int row_start=row_offsets[i];
32      int row_end=row_offsets[i+1];
33      for(int j=row_start;j<row_end;j++) {
34        unsigned int Acol=cols[j];
35        double Acoef=Acoefs[j];
36        double xcoef=xcoefs[Acol];
37        sum+=Acoef*xcoef;
38      }
39      ycoefs[i]=sum;
40    }
41  }
```

Benchmark Code

Building the Conjugate Gradient Code - **Makefile**

- It is important to compile using the **-O0** to have a proper profilling output;
- **ta** option provides informations about which architecture the code wiil be run. Some options are: **host, tesla and multicore**;
- **ccff** stands for *Common Compiler Feedback Framework*. It provides additional to the executable that can be used by the **PGPROF** utility to display informations about how the compiler optimized the code.

```
1  CXX=pgc++
2  CXXFLAGS=-O0 -Minfo=all,ccff -Mneginfo
3  LDFLAGS=${CXXFLAGS}
4
5  cg.x: main.o
6          ${CXX} $^ -o $@ ${LDFLAGS}
7  main.o: main.cpp matrix.h matrix_functions.h vector.h vector_functions.h
8
9  .SUFFIXES: .o .cpp .h
10
11 .PHONY: clean
12
13 clean:
14         rm -Rf cg.x pgprof* core *.o
```

Benchmark Code

Expected Output

- The exact time required to run the executable will vary depending on the architecture that the code will be run run along with the parallelism level implemented in the code.
- The tolerance for each iteration shall remain the same as presented in following output:

```
Rows: 3442951, nnz: 92553775
Iteration: 0, Tolerance: 2.8121e+08
Iteration: 10, Tolerance: 1.3910e+07
Iteration: 20, Tolerance: 4.7619e+05
Iteration: 30, Tolerance: 1.7089e+04
Iteration: 40, Tolerance: 6.1091e+02
Iteration: 50, Tolerance: 2.1671e+01
Iteration: 60, Tolerance: 7.5519e-01
Iteration: 70, Tolerance: 2.4847e-02
Iteration: 80, Tolerance: 5.8162e-05
Iteration: 90, Tolerance: 2.5592e-06
Total Iterations: 100 Total Time: 29.762435s
Final Tolerance: 1.2610e-07
```

Benchmark Code

Initial Benchmarking

- PGProf profiler is used in the present example;
- The first step is to profile the serial version of the code to understand where the executable is spending its time so that one can focus our efforts on the functions and loops that will deliver the greatest performance;
- PGI compiler 19.10 does not provide a graphical time line for applications without OpenACC directives;
- One can use the line command line to profile the code without the graphical interface: **$pgprof ./cg.x**

```
1  ======== CPU profiling result (bottom up):
2  Time(%)      Time  Name
3   87.43%    26.58s  matvec(matrix const &, vector const &, vector const &)
4   87.43%    26.58s  | main
5    6.51%     1.98s  waxpby(double, vector const &, double, vector const &, vector const &)
6    6.51%     1.98s  | main
7    4.87%     1.48s  dot(vector const &, vector const &)
8    4.87%     1.48s  | main
9    1.12%     340ms  allocate_3d_poisson_matrix(matrix&, int)
10   1.12%     340ms  | main
11   0.03%      10ms  munmap
12   0.03%      10ms  | free_vector(vector&)
13   0.03%      10ms  |   main
14   0.03%      10ms  initialize_vector(vector&, double)
15   0.03%      10ms    main
16
17 ======== Data collected at 100Hz frequency
```

Benchmark Code

Initial Benchmarking

- One can also use **Perf** profiler tool for the sequential analysis:
  - **Perf** is a profiler tool for Linux 2.6+ based systems that abstracts away CPU hardware differences in Linux performance measurements and presents a simple command line interface.

- Record data:

  $ perf record -g ./cg.x

- Report data:

  $ perf report

```
Samples: 123K of event 'cycles:ppp', Event count (approx.): 132790212791
  Children      Self  Command  Shared Object      Symbol
-   87.52%    87.39%  cg.x     cg.x               [.] matvec
     matvec
-    6.50%     6.48%  cg.x     cg.x               [.] waxpby
     waxpby
-    4.82%     4.81%  cg.x     cg.x               [.] dot
     dot
-    1.10%     0.95%  cg.x     cg.x               [.] allocate_3d_poisson_matrix
     allocate_3d_poisson_matrix
```

Benchmark Code

Initial Benchmarking

- There are three functions of interest:
  1. **matvec** ($\approx 87.5\%$): doubly nested loop that performs a sparse matrix/vector product;
  2. **waxpy** ($\approx 6.5\%$): single loop that performs $aX + bY$ vector operation;
  3. **dot** ($\approx 4.8\%$): single loop that performs a dot product;
- **allocate_3d_poisson_matrix** ($\approx 1\%$) is an initialization routine that is only run once. Therefore, it is ignored in the present optimization;
- It is generally a best practice to start the introduction of OpenACC directives at the top of the time consuming routines and work the way down;
- As an educational approach we are starting from the less complex function and work our way up.

Accelerating WAXPBY

First Step

- The first step is the addition of the OpenACC **kernel** pragma in this one loop function along with enabling the use of the OpenACC directives, for NVIDIA GPUs, using *-ta=tesla* command-line, into the Makefile.

```
33 void waxpby(double alpha, const vector &x, double beta, const vector &y, const vector& w) {
34   unsigned int n=x.n;
35   double *xcoefs=x.coefs;
36   double *ycoefs=y.coefs;
37   double *wcoefs=w.coefs;
38
39 #pragma acc kernels
40   {
41     for(int i=0;i<n;i++) {
42       wcoefs[i]=alpha*xcoefs[i]+beta*ycoefs[i];
43     }
44   }
45 }
```

```
CXXFLAGS=-O0 -ta=tesla,lineinfo -Minfo=all,ccff -Mneginfo
```

Accelerating WAXPBY

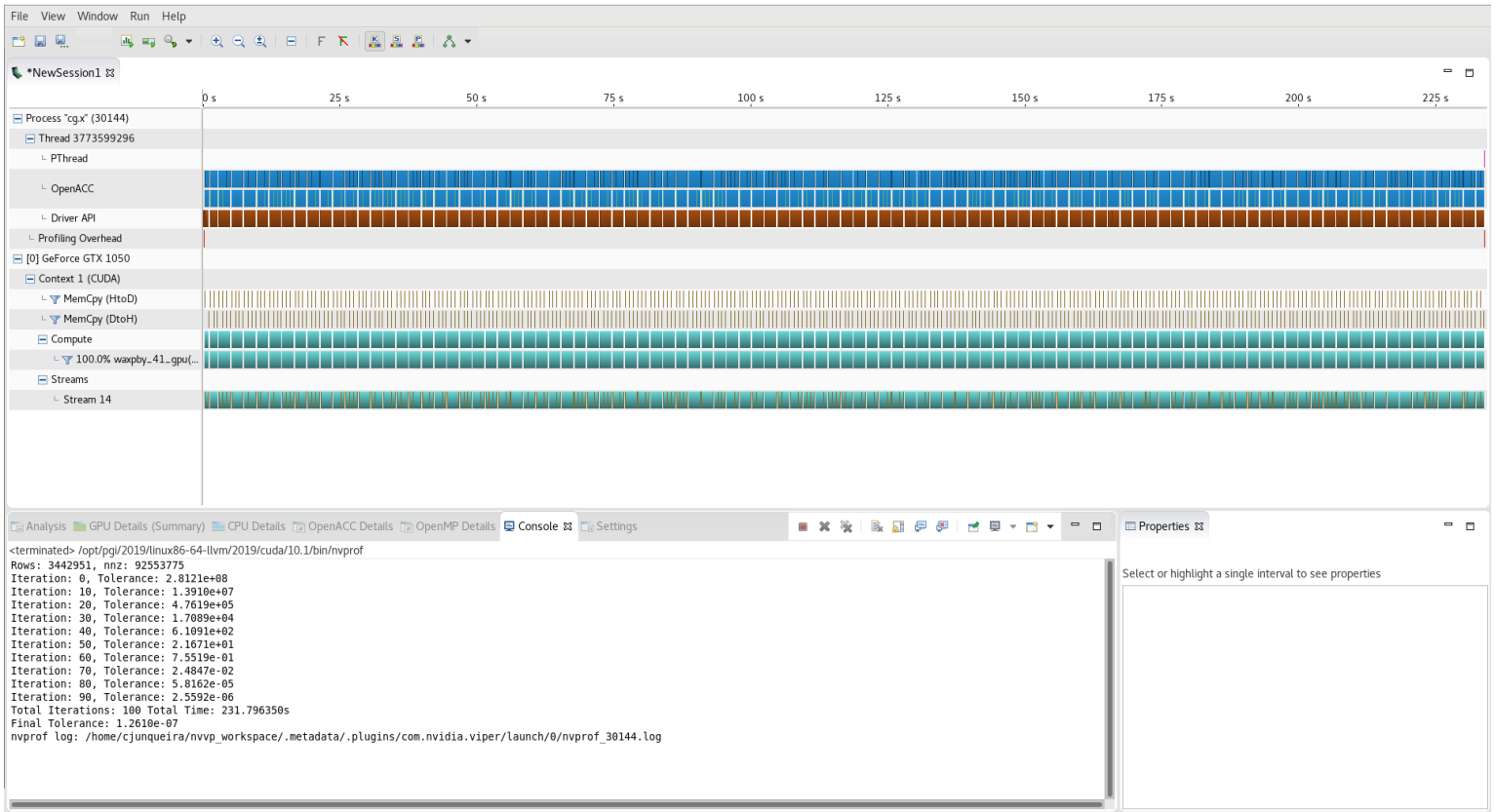
First Step

- The compiler has identified a possible data dependency between interations of the loop;
- C and C++ programming languages represent arrays simply as pointers into memory. These pointers can potentially point to the same memory;
- The compiler cannot prove that the three arrays used in the loop do not alias, or everlap, which each other. Hence, it has assumed that parallelizing the loop would be unsafe.

```
pgc++ -O0 -ta=tesla,lineinfo -Minfo=all,ccff -Mneginfo main.cpp -o cg.x
dot(const vector &, const vector &):
     31, FMA (fused multiply-add) instruction(s) generated
waxpby(double, const vector &, double, const vector &, const vector &):
     21, include "vector_functions.h"
          40, Generating implicit copyin(ycoefs[:n],xcoefs[:n]) [if not already present]
              Generating implicit copyout(wcoefs[:n]) [if not already present]
          41, Complex loop carried dependence of ycoefs-> prevents parallelization
              Loop carried dependence of wcoefs-> prevents parallelization
              Loop carried backward dependence of wcoefs-> prevents vectorization
              Accelerator serial kernel generated
              Generating Tesla code
              41, #pragma acc loop seq
matvec(const matrix &, const vector &, const vector &):
     21, include "vector_functions.h"
          41, FMA (fused multiply-add) instruction(s) generated
```

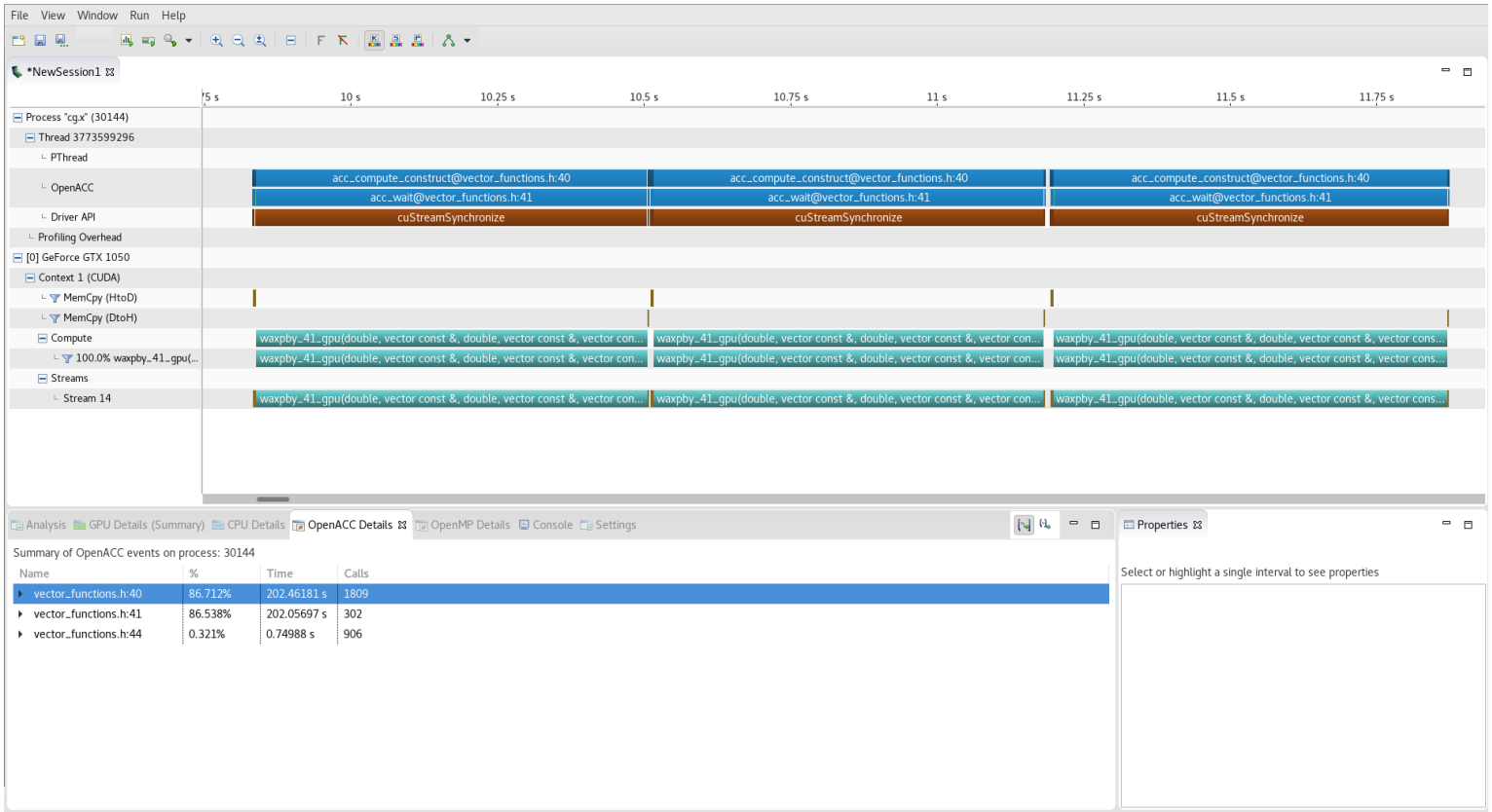
Accelerating WAXPBY

First Step



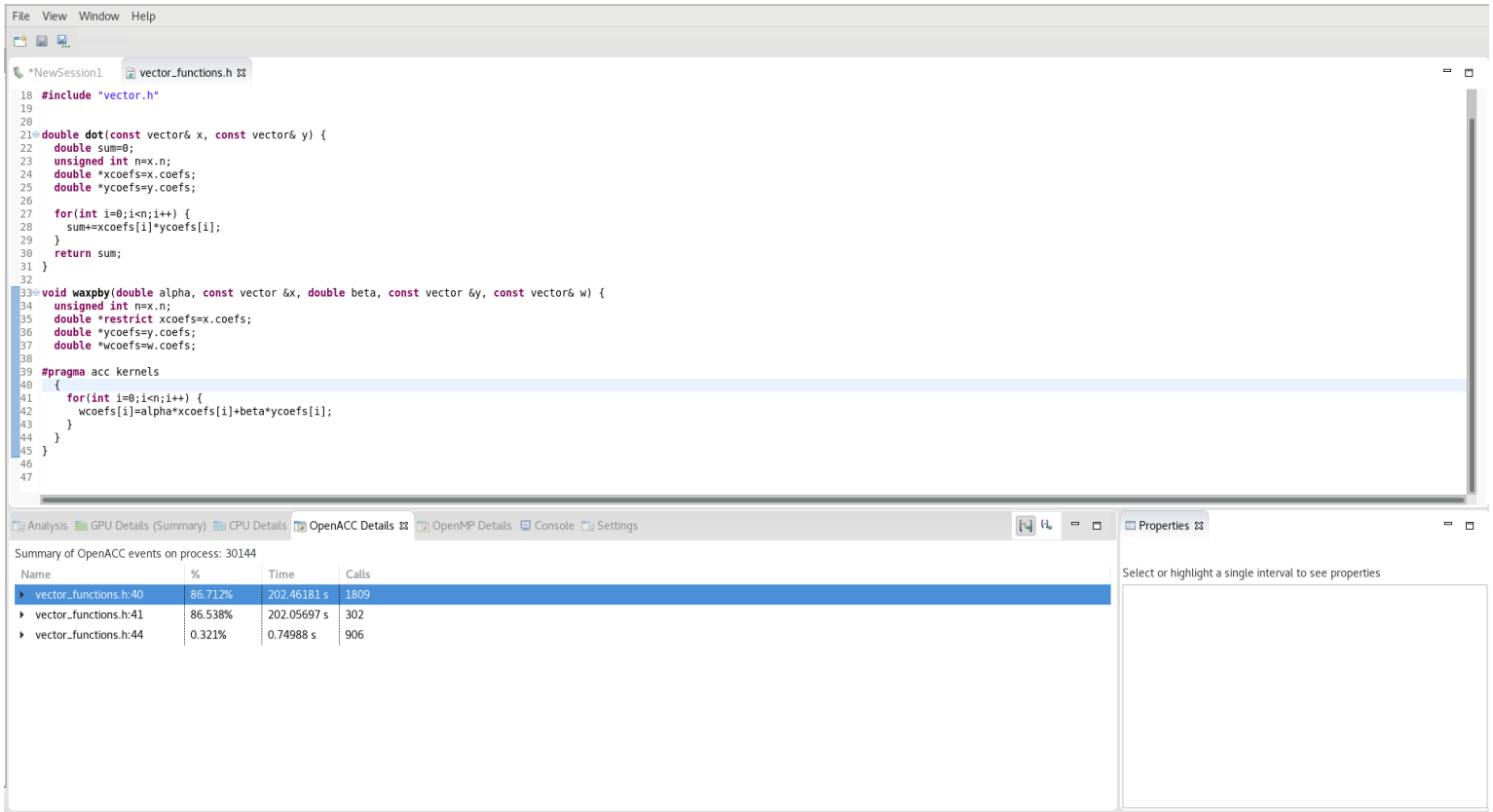
Accelerating WAXPBY

First Step



Accelerating WAXPBY

First Step



Accelerating WAXPBY

Second Step

- One can provide the compiler more information about the loop itself using OpenACC loop directive along with the independent clause;
- The independent clause informs the OpenACC compiler that no two iterations of the loop depend up the data of the other, thus overriding the compiler dependency analysis of the loop;
- The choice is a promise that the programmer makes to the compiler about the code. Unpredictable results can occur once such promise is broken.

```
33 void waxpby(double alpha, const vector &x, double beta, const vector &y, const vector& w) {
34   unsigned int n=x.n;
35   double *xcoefs=x.coefs;              //double *restrict xcoefs=x.coefs;
36   double *ycoefs=y.coefs;              //restrict (C fashion) or __restrict (C++ fashion)
37   double *wcoefs=w.coefs;              //can be used to promise to the compiler
38                                        //that the pointers will never alias
39 #pragma acc kernels copyout(wcoefs[:w.n]) copyin(ycoefs[:y.n],xcoefs[:x.n])
40   { //copy directives are added as suggested by the compiler
41     #pragma acc loop independent                    //the independent clause tell the
42     for(int i=0;i<n;i++) {                          //compiler that there is no data
43       wcoefs[i]=alpha*xcoefs[i]+beta*ycoefs[i];     //dependency in this loop
44     }
45   }
```

Accelerating WAXPBY

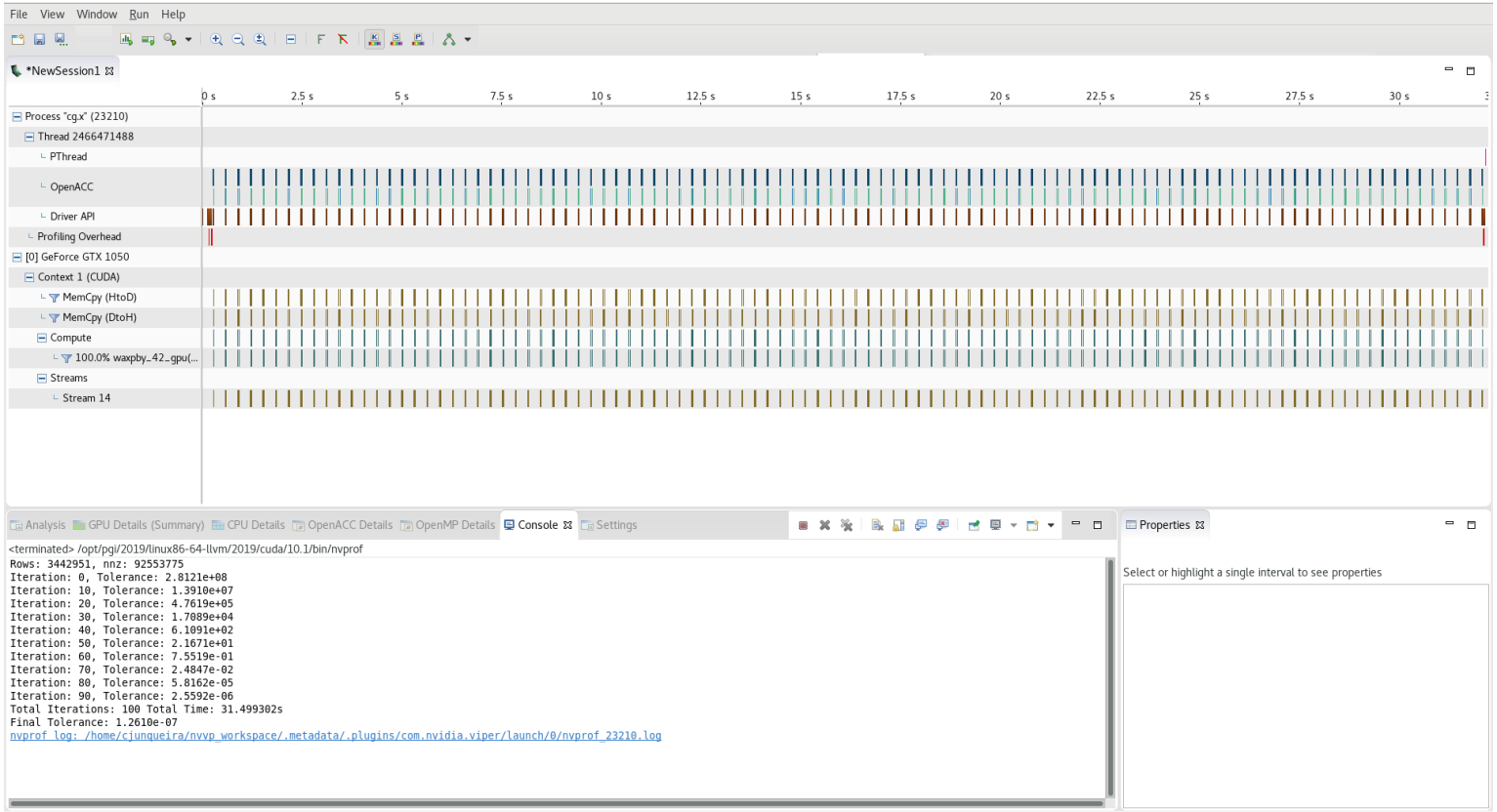
Second Step

- The compiler has now determined that the loop at line 42 is parallelizable.

```
pgc++ -O0 -ta=tesla,lineinfo -Minfo=all,ccff -Mneginfo main.cpp -o cg.x
dot(const vector &, const vector &):
     31, FMA (fused multiply-add) instruction(s) generated
waxpby(double, const vector &, double, const vector &, const vector &):
     21, include "vector_functions.h"
          40, Generating copyin(xcoefs[:n],ycoefs[:n]) [if not already present]
              Generating copyout(wcoefs[:n]) [if not already present]
          42, Loop is parallelizable
              Generating Tesla code
              42, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
matvec(const matrix &, const vector &, const vector &):
     21, include "vector_functions.h"
          41, FMA (fused multiply-add) instruction(s) generated
```

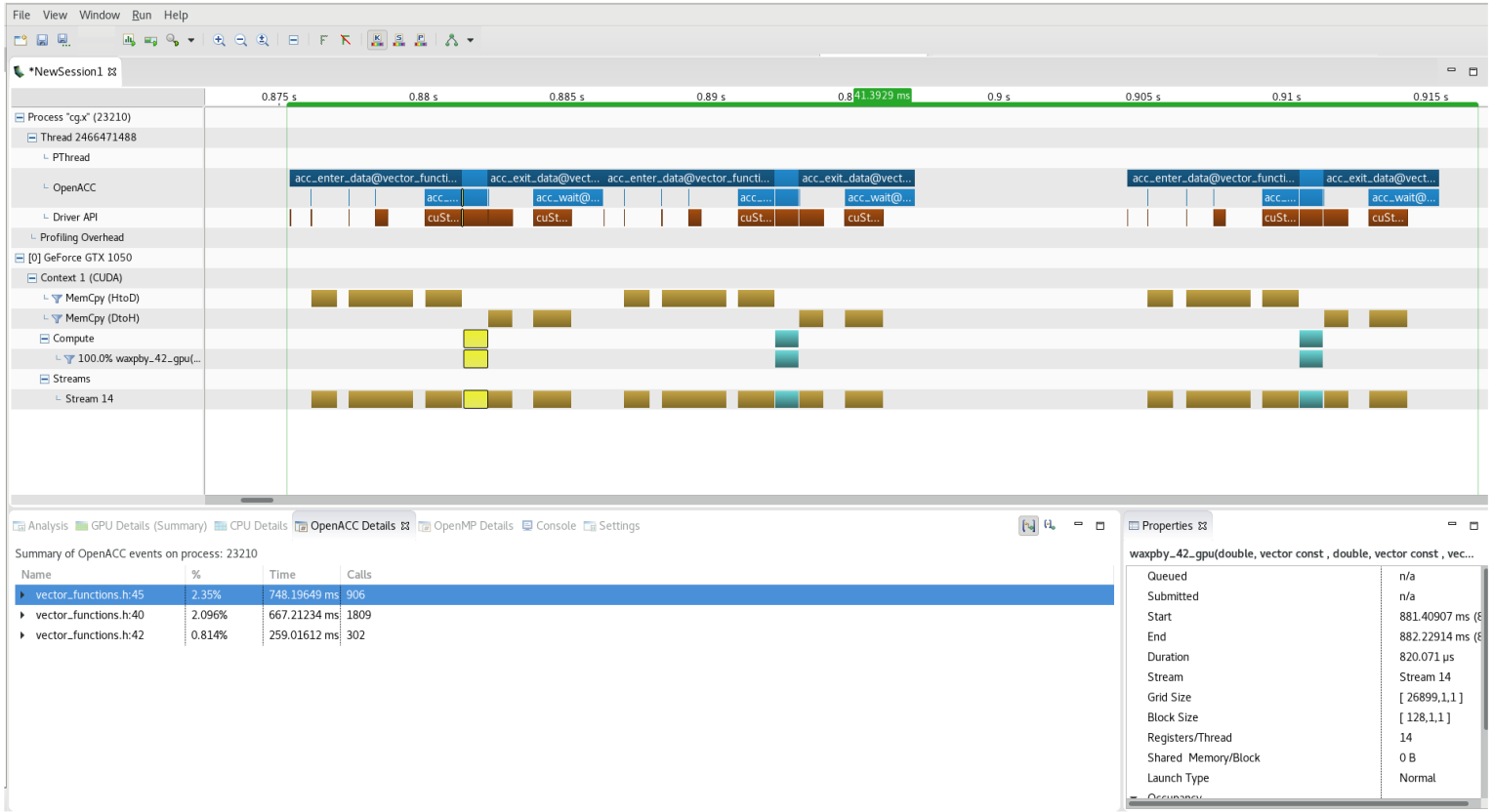
Accelerating WAXPBY

Second Step



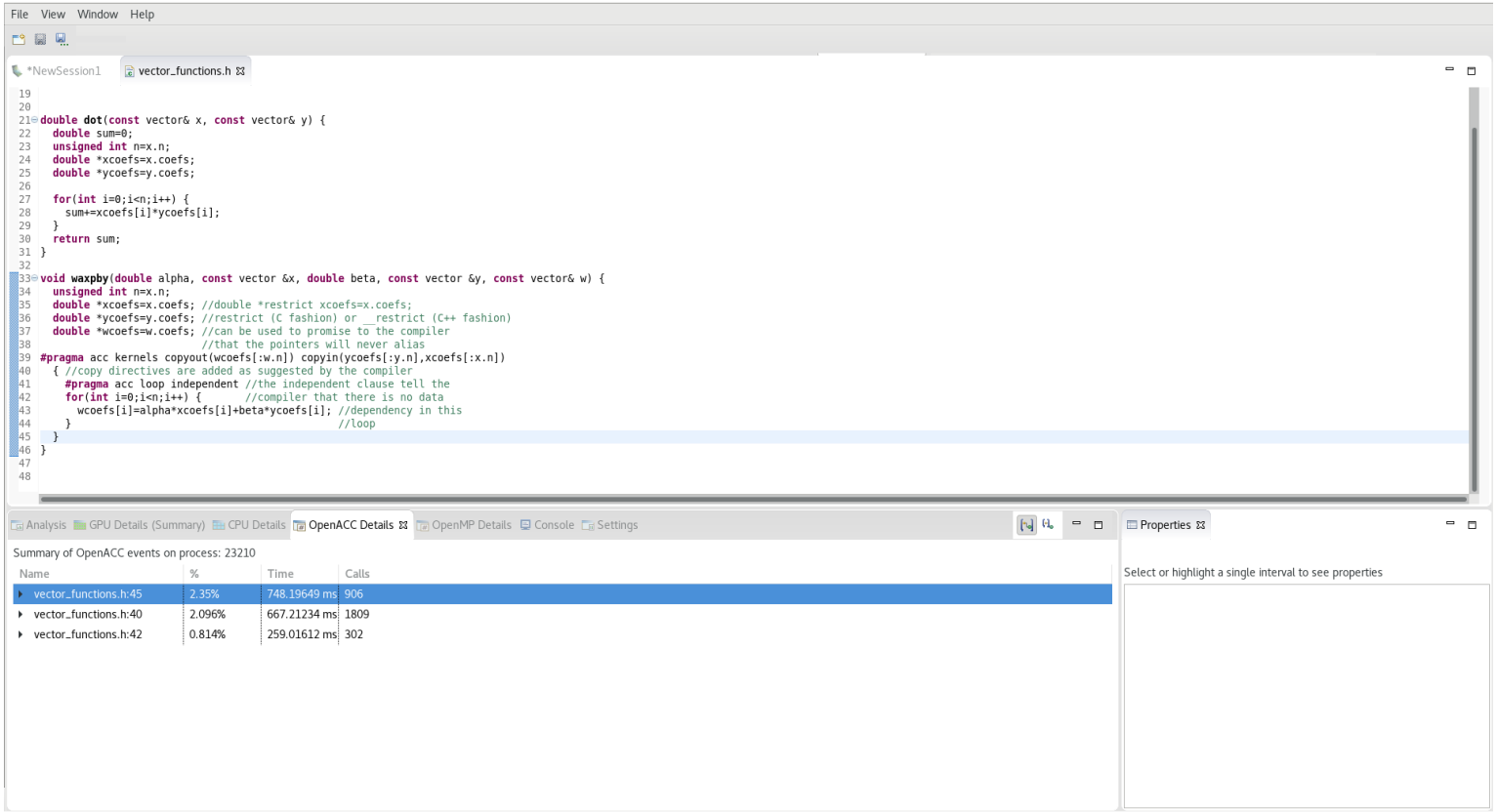
Accelerating WAXPBY

Second Step



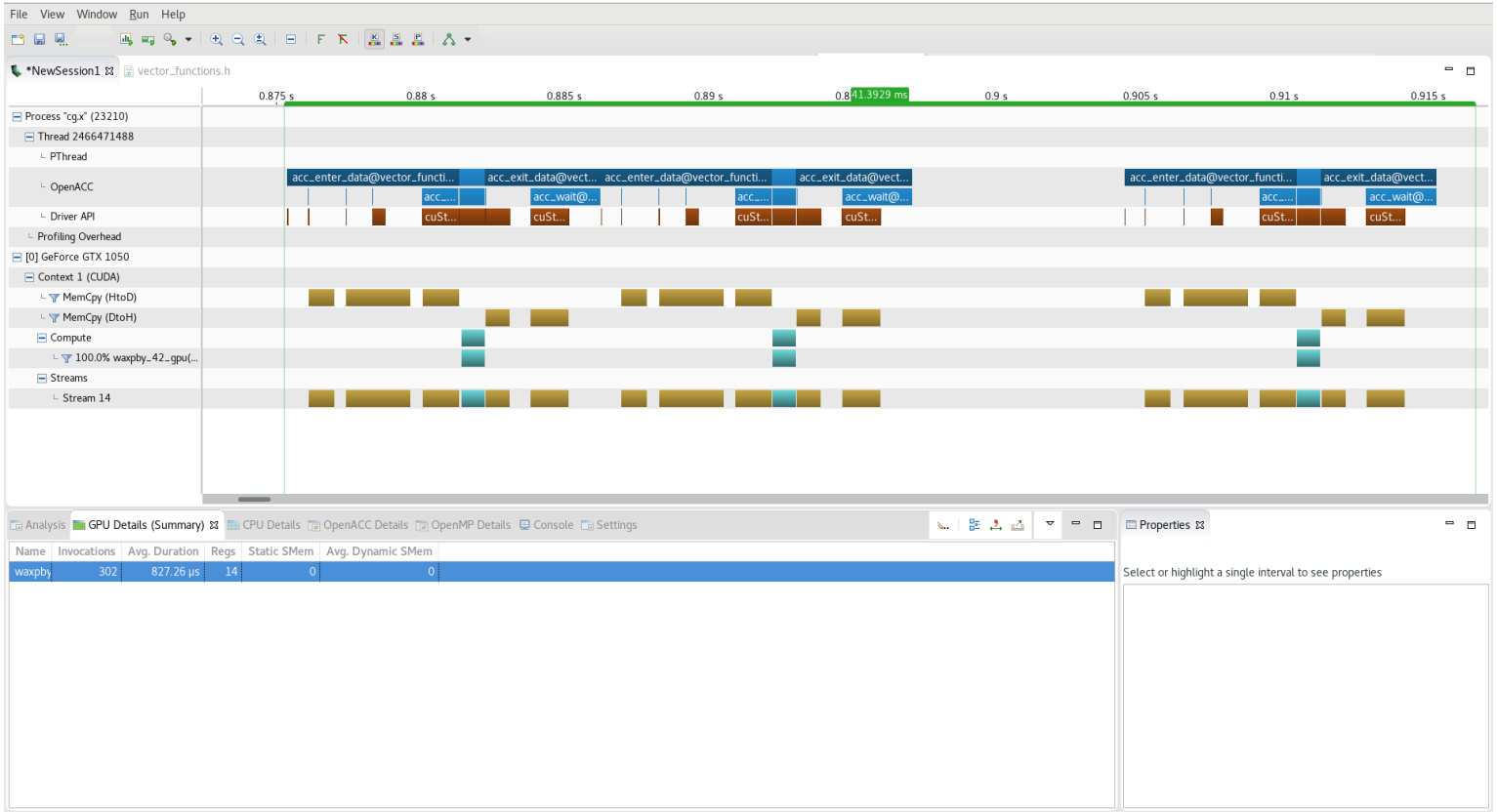
Accelerating WAXPBY

Second Step



Accelerating WAXPBY

Second Step



Accelerating WAXPBY

Second Step

- The code has slowed down when comparing the total time of the accelerated WAXPBY with the sequential version;
- For each call to WAXPBY there is a copy of two arrays to the device and one array back for use by other function, which is efficient;
- The ideal configuration is to leave data in GPU memory for as long as possible and avoid intermediate data transfers. The only way to do that is to accelerate the remaining functions so that the data transfers become unnecessary.

Accelerating DOT

OpenACC Kernel pragma

- One can add the OpenACC kernel pragma to the one loop dot function.

```
21 double dot(const vector& x, const vector& y) {
22   double sum=0;
23   unsigned int n=x.n;
24   double *xcoefs=x.coefs;
25   double *ycoefs=y.coefs;
26
27 #pragma acc kernels
28   for(int i=0;i<n;i++) {
29     sum+=xcoefs[i]*ycoefs[i];
30   }
31   return sum;
32 }
```

Accelerating DOT

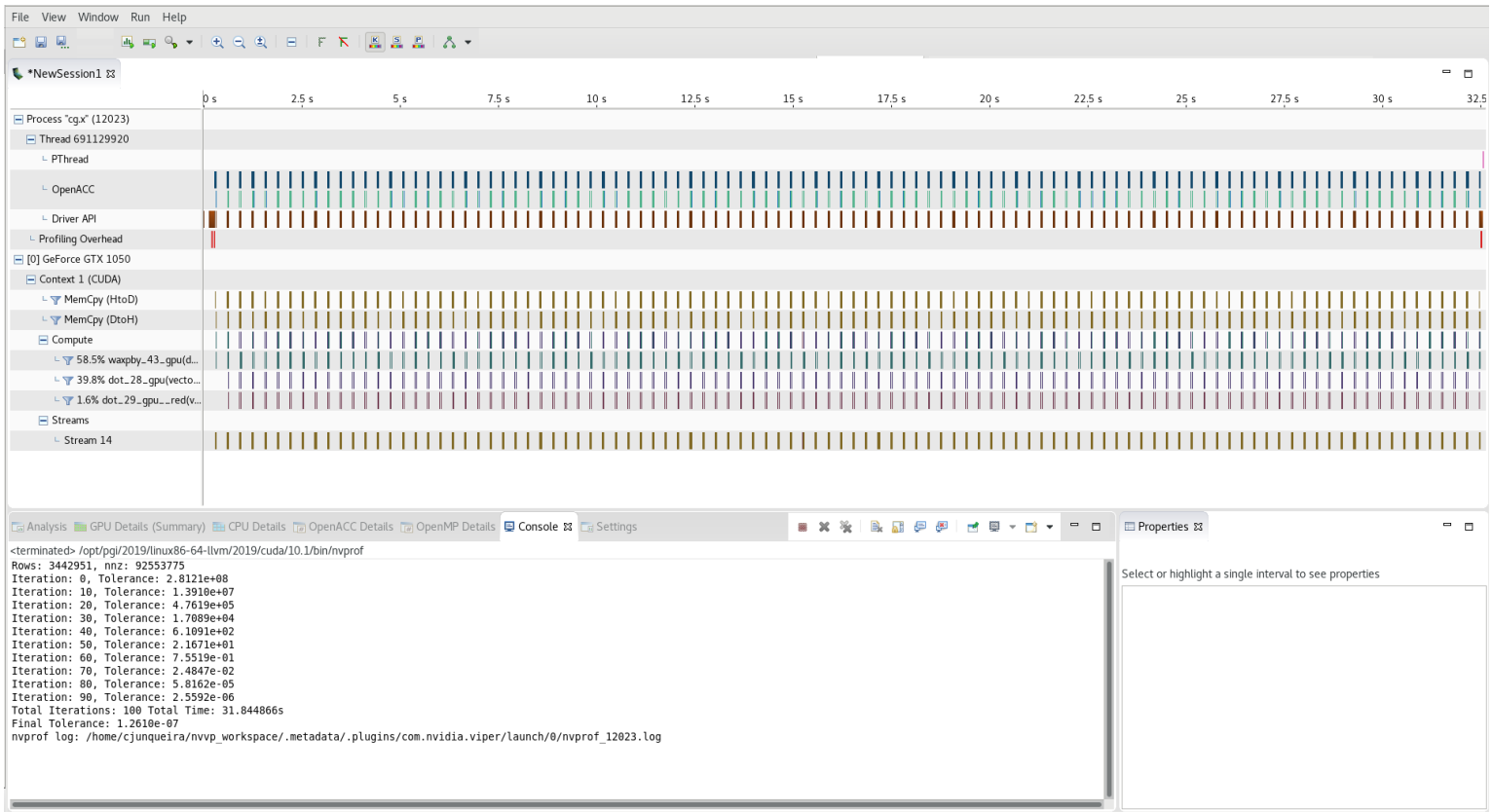
OpenACC Compiler Feedback

- The compiler parallelized the loop in line 28;
- The compiler handled the complexity of identifying the *reduction* in line 29 because of the use of the kernels pragma to parallelize the loop within dot function;
- The programmer is responsible to inform the compiler of the reduction when using the more advanced OpenACC *parallel* pragma.

```
pgc++ -O0 -ta=tesla,lineinfo -Minfo=all,ccff -Mneginfo main.cpp -o cg.x
dot(const vector &, const vector &):
     21, include "vector_functions.h"
          25, Generating implicit copyin(xcoefs[:n],ycoefs[:n]) [if not already present]
          28, Loop is parallelizable
              Generating Tesla code
              28, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
              29, Generating implicit reduction(+:sum)
waxpby(double, const vector &, double, const vector &, const vector &):
     21, include "vector_functions.h"
          41, Generating copyin(xcoefs[:x->n]) [if not already present]
              Generating copy(ycoefs[:y->n],wcoefs[:w->n]) [if not already present]
          43, Loop is parallelizable
              Generating Tesla code
              43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
matvec(const matrix &, const vector &, const vector &):
     21, include "vector_functions.h"
          41, FMA (fused multiply-add) instruction(s) generated
```

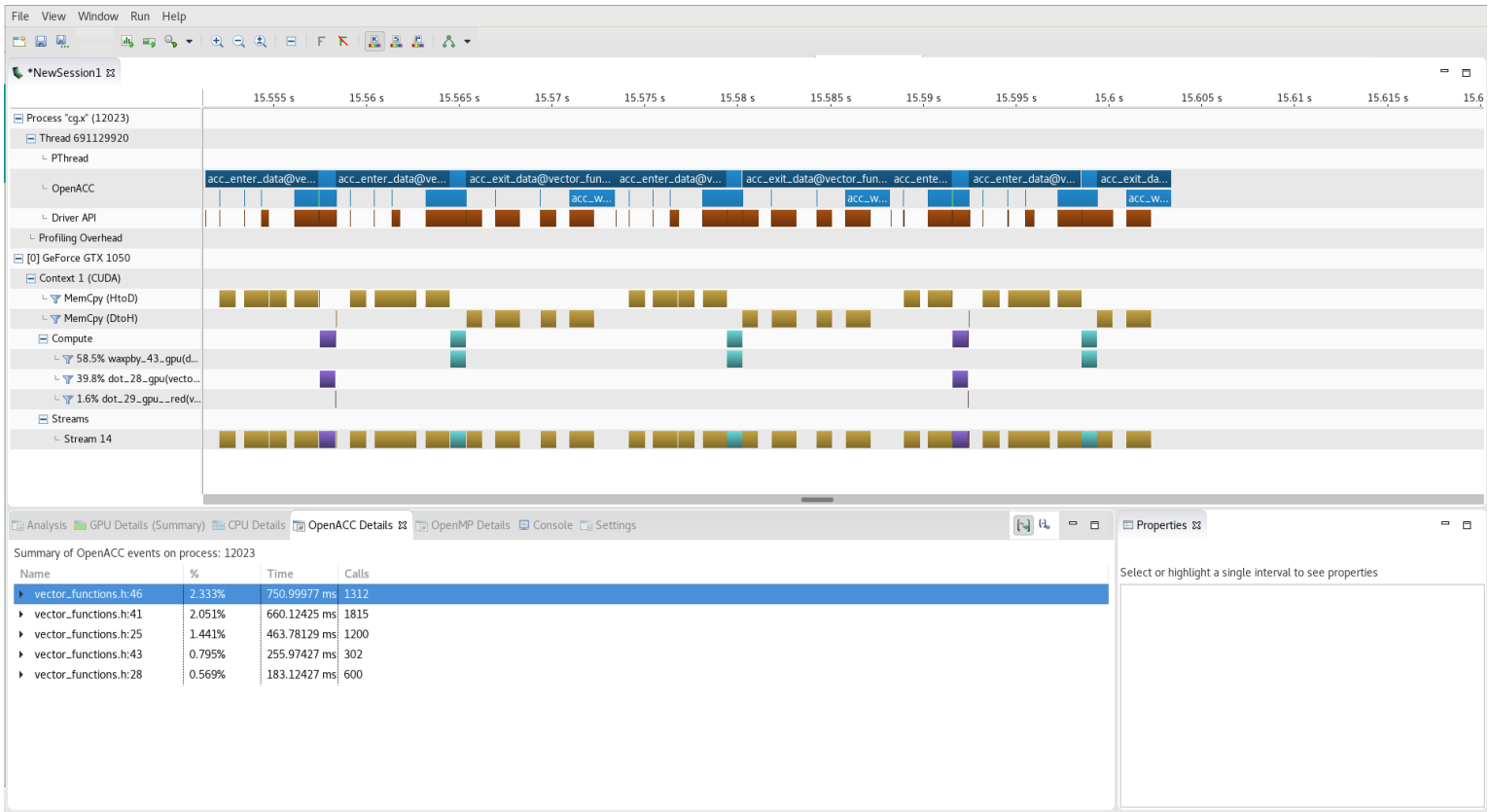
Accelerating DOT

Profile Timeline



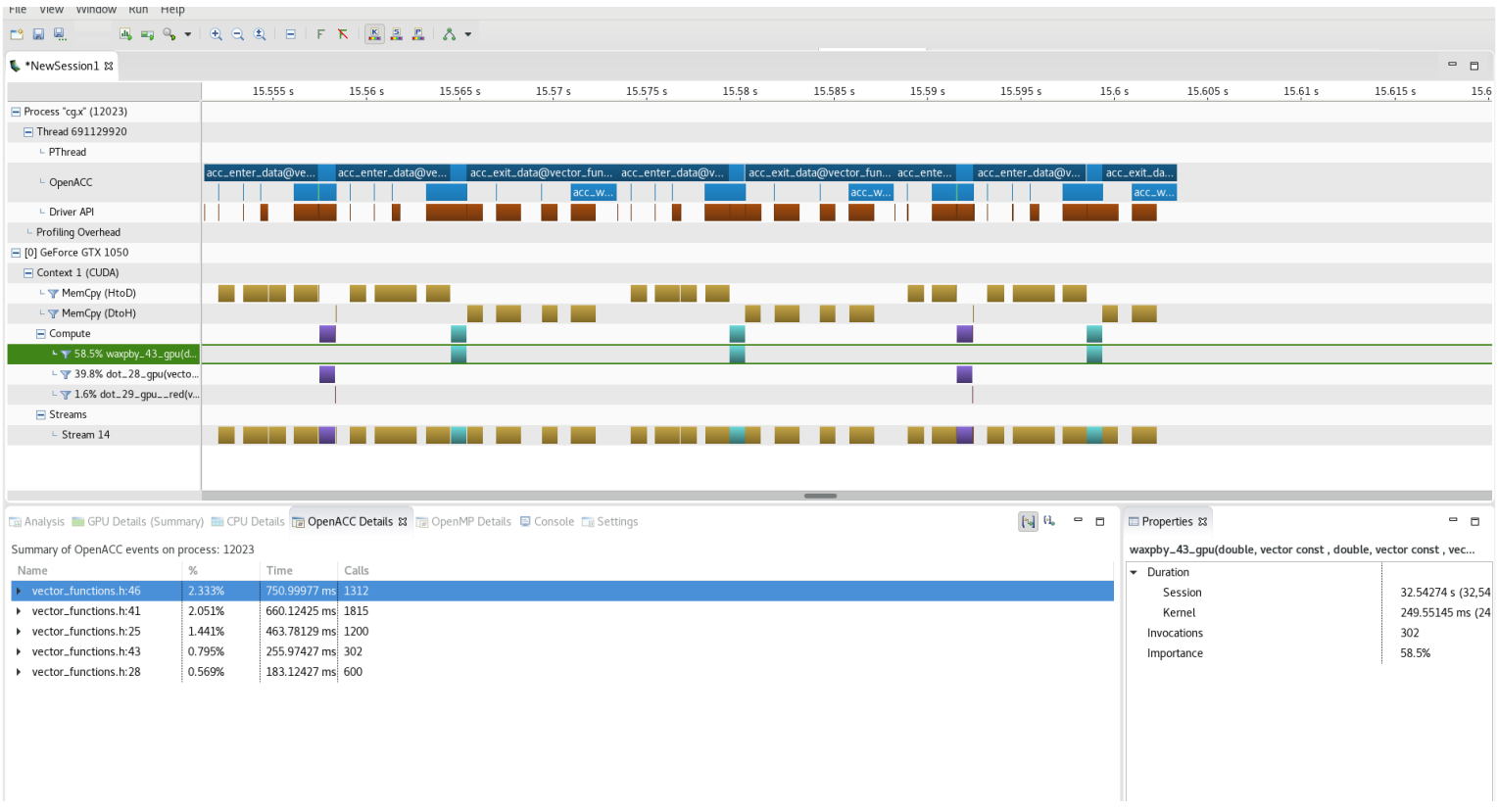
Accelerating DOT

Profile Timeline



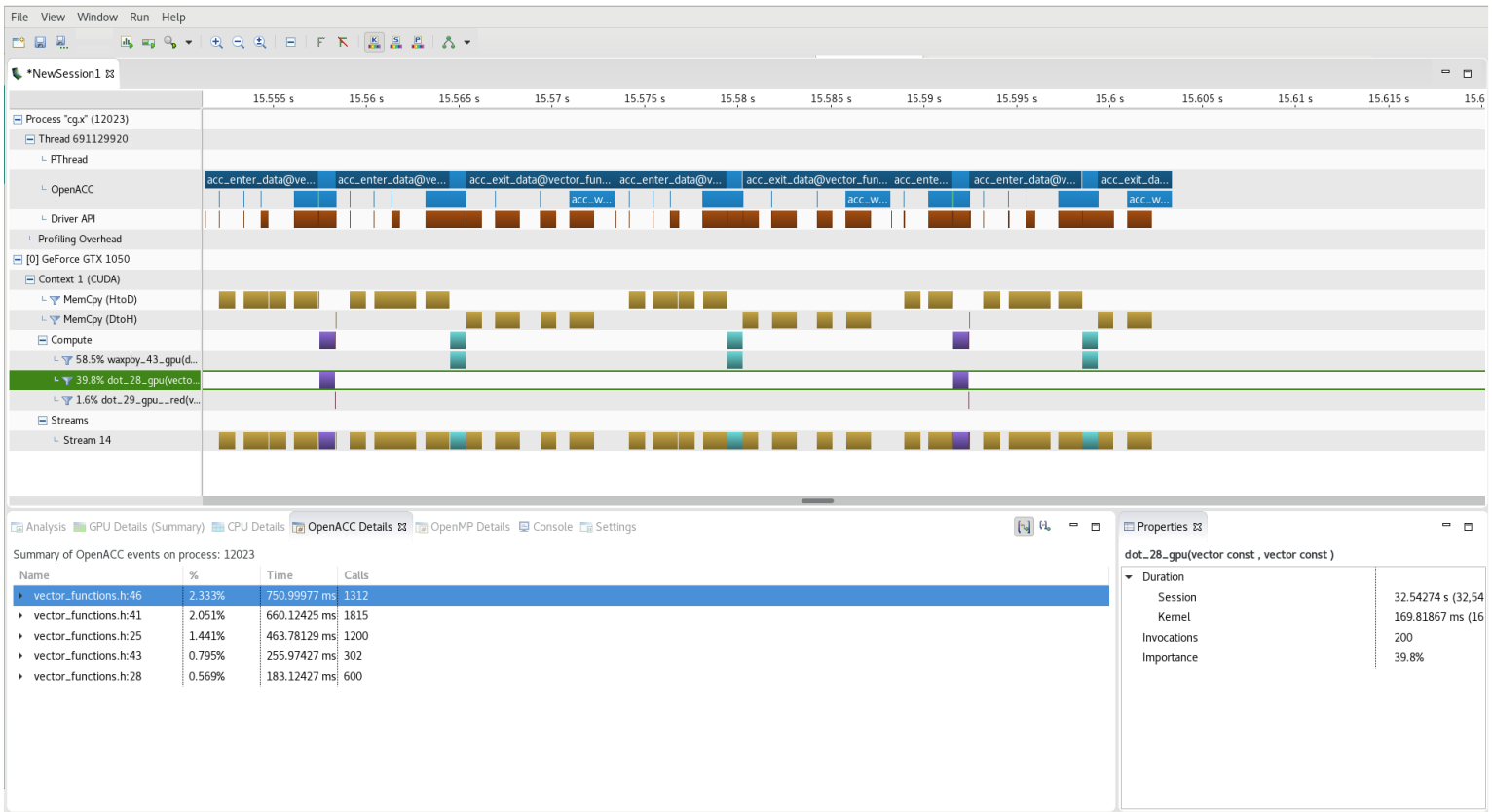
Accelerating DOT

Profile Timeline



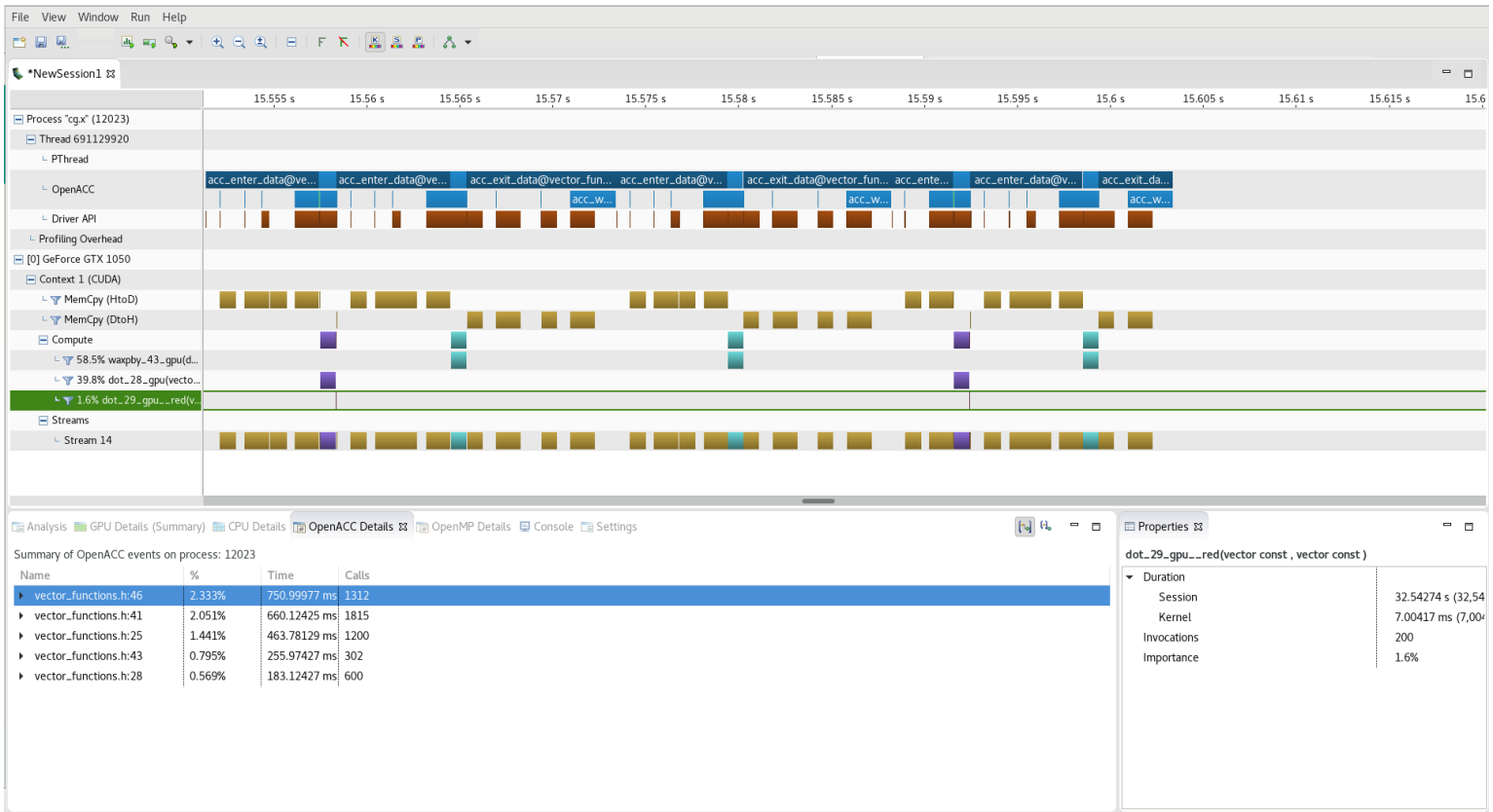
Accelerating DOT

Profile Timeline



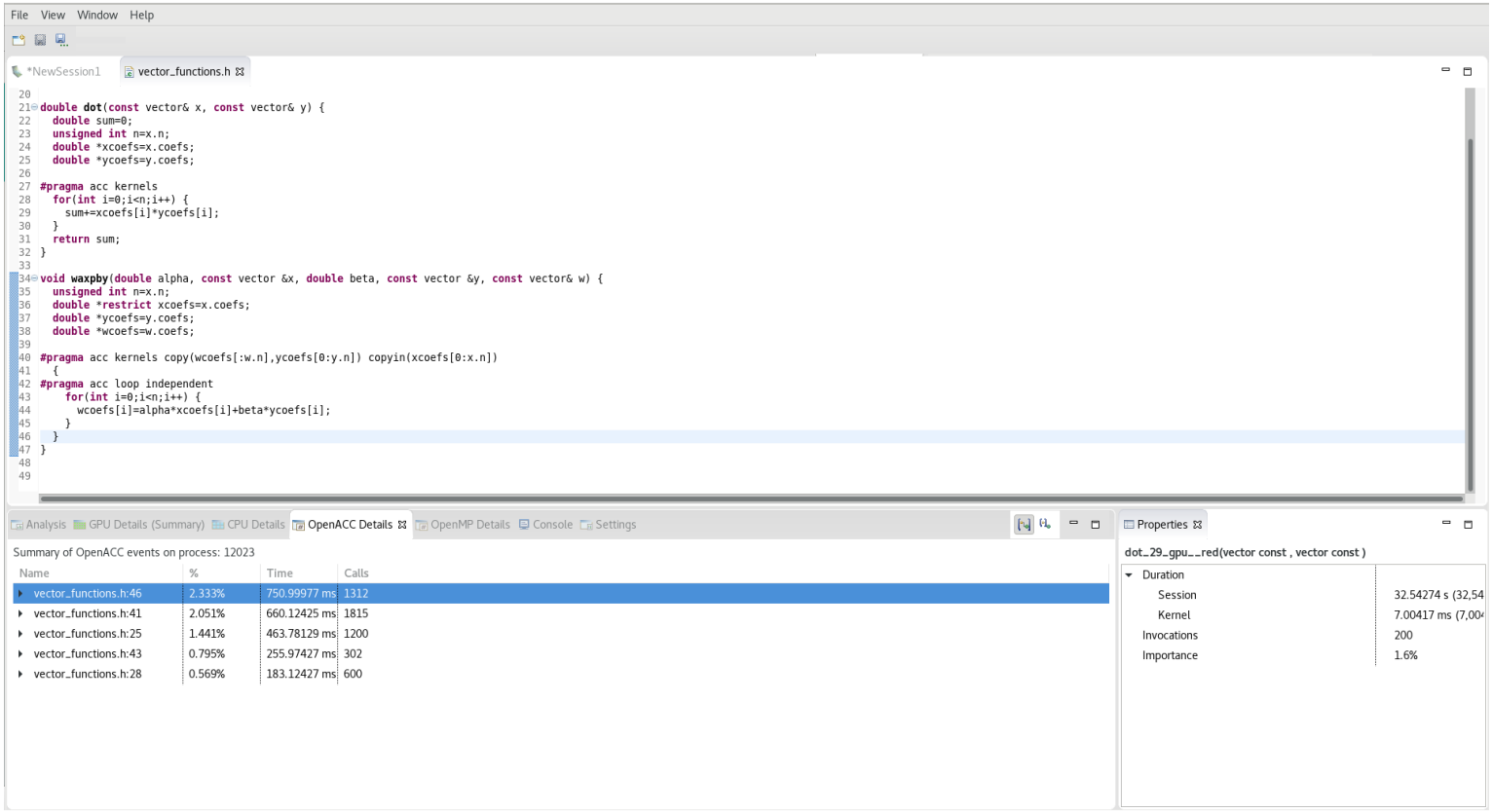
Accelerating DOT

Profile Timeline



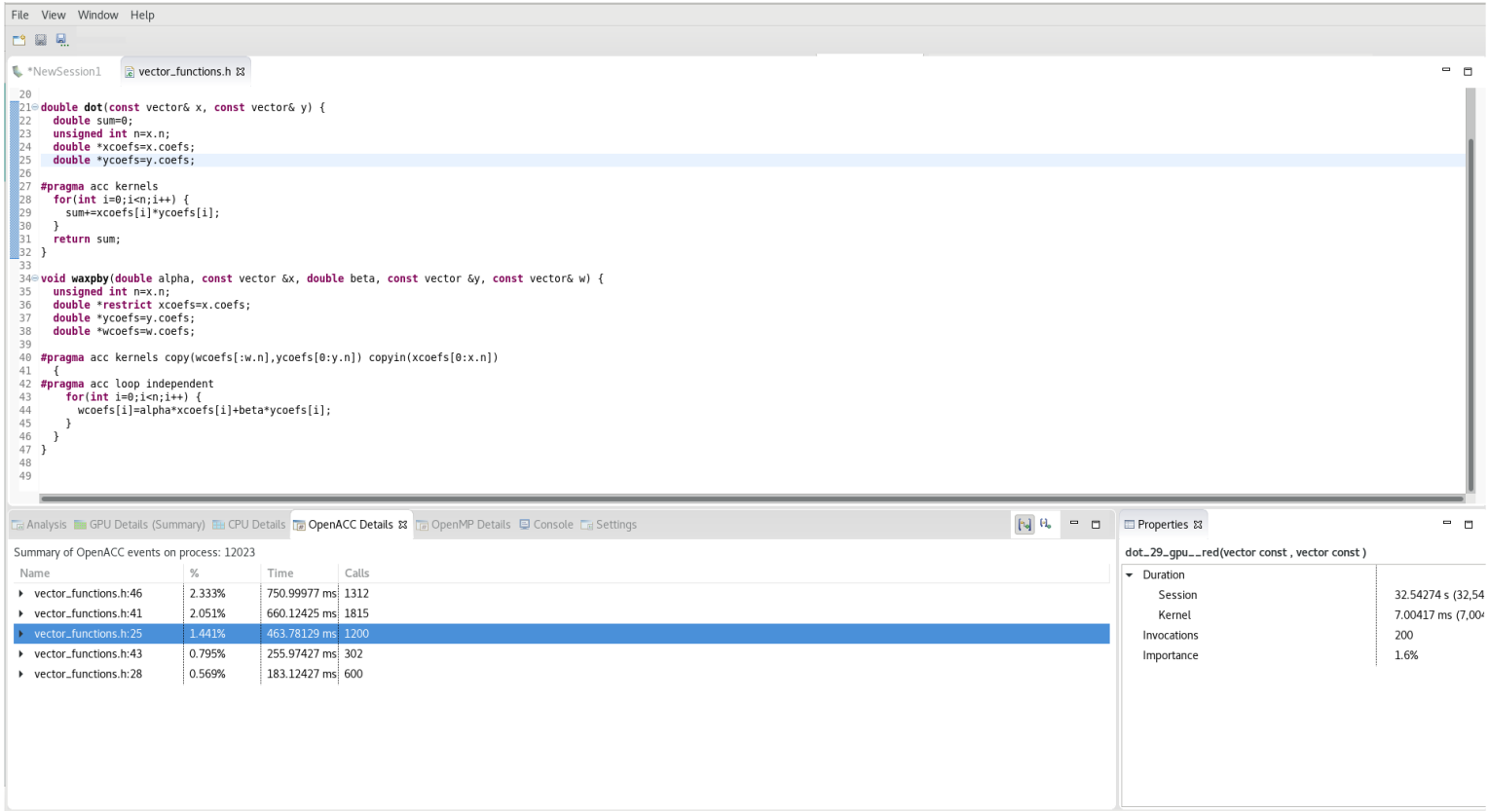
Accelerating DOT

Profile Timeline



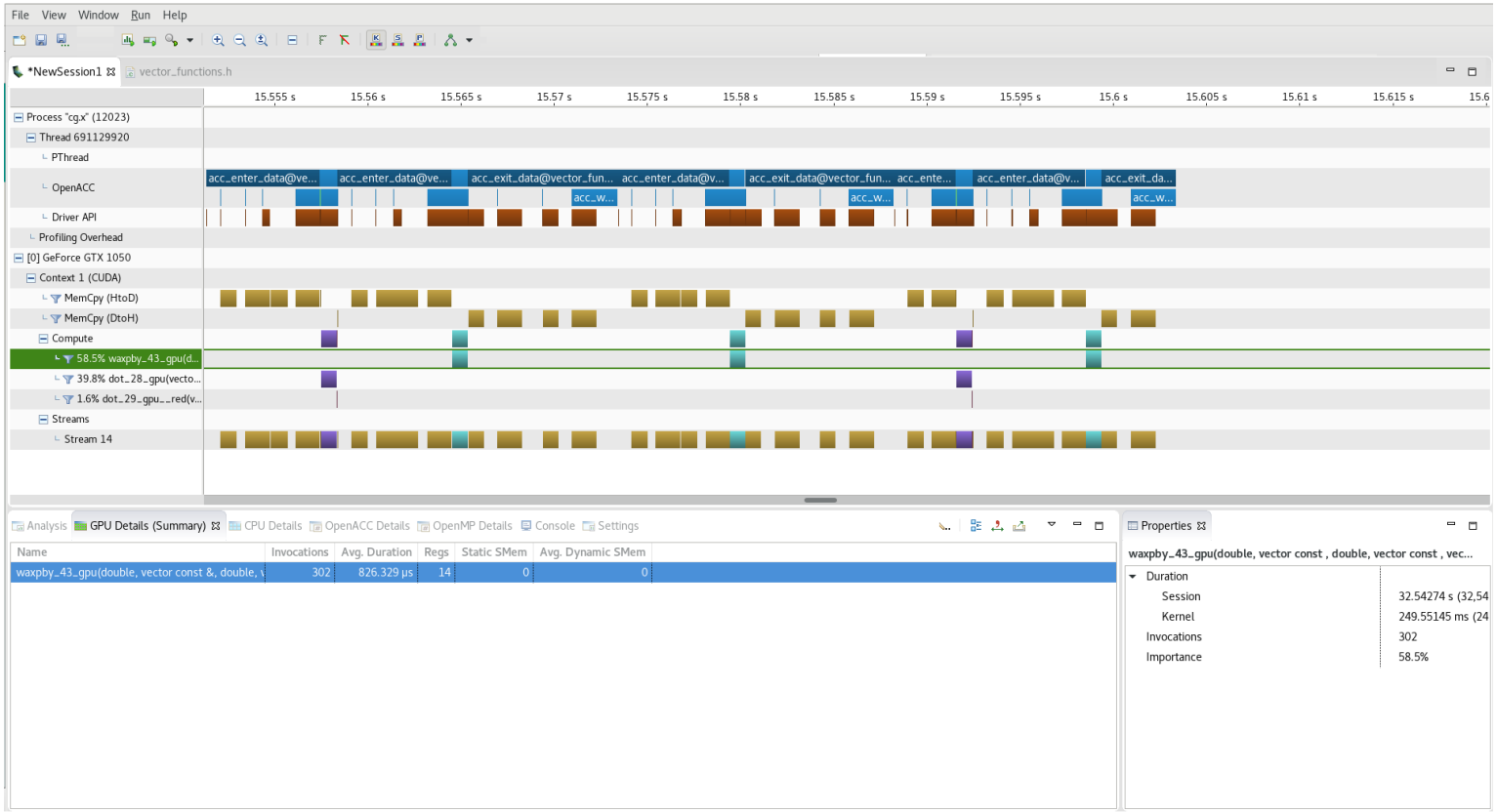
Accelerating DOT

Profile Timeline



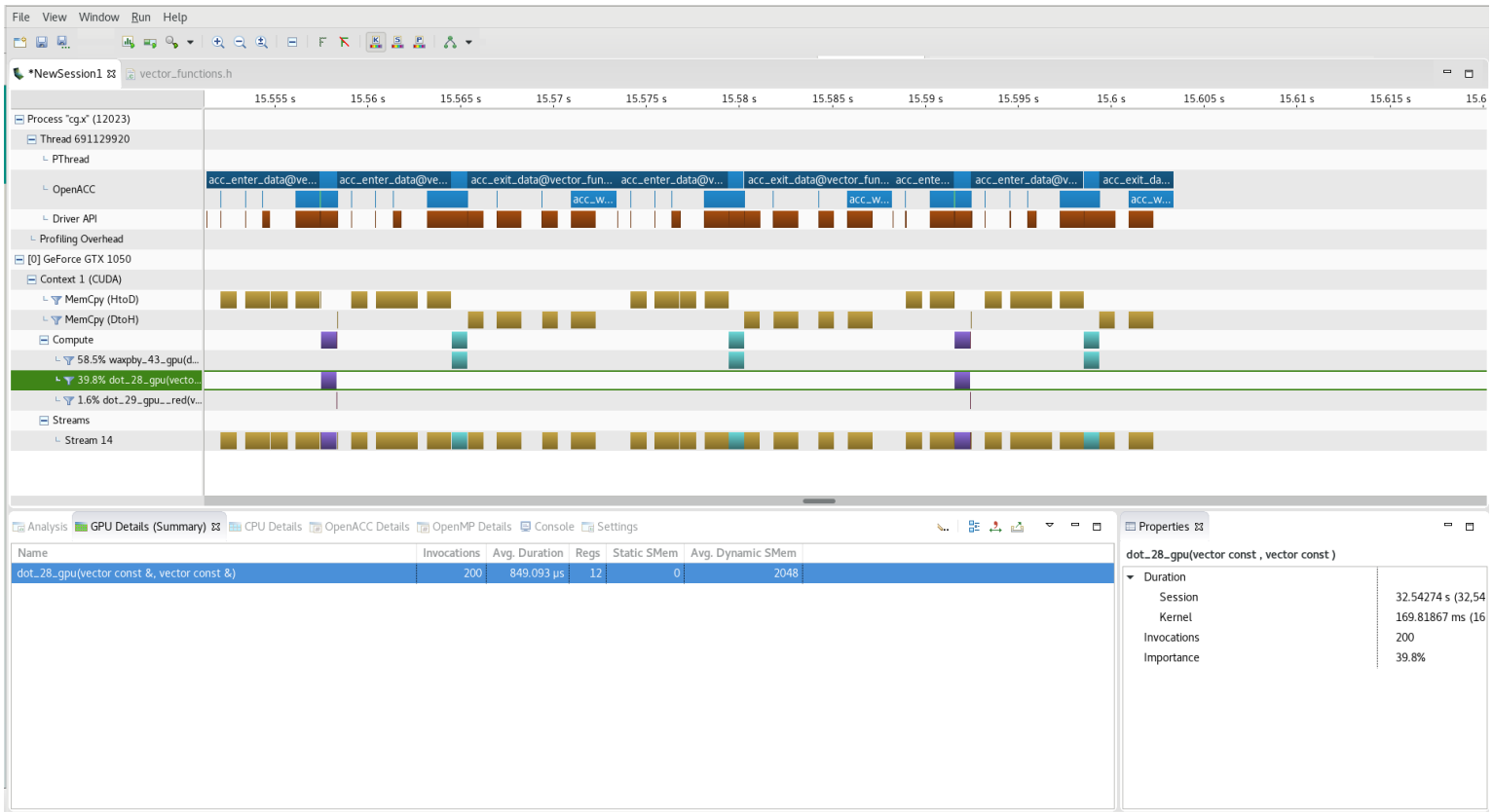
Accelerating DOT

Profile Timeline



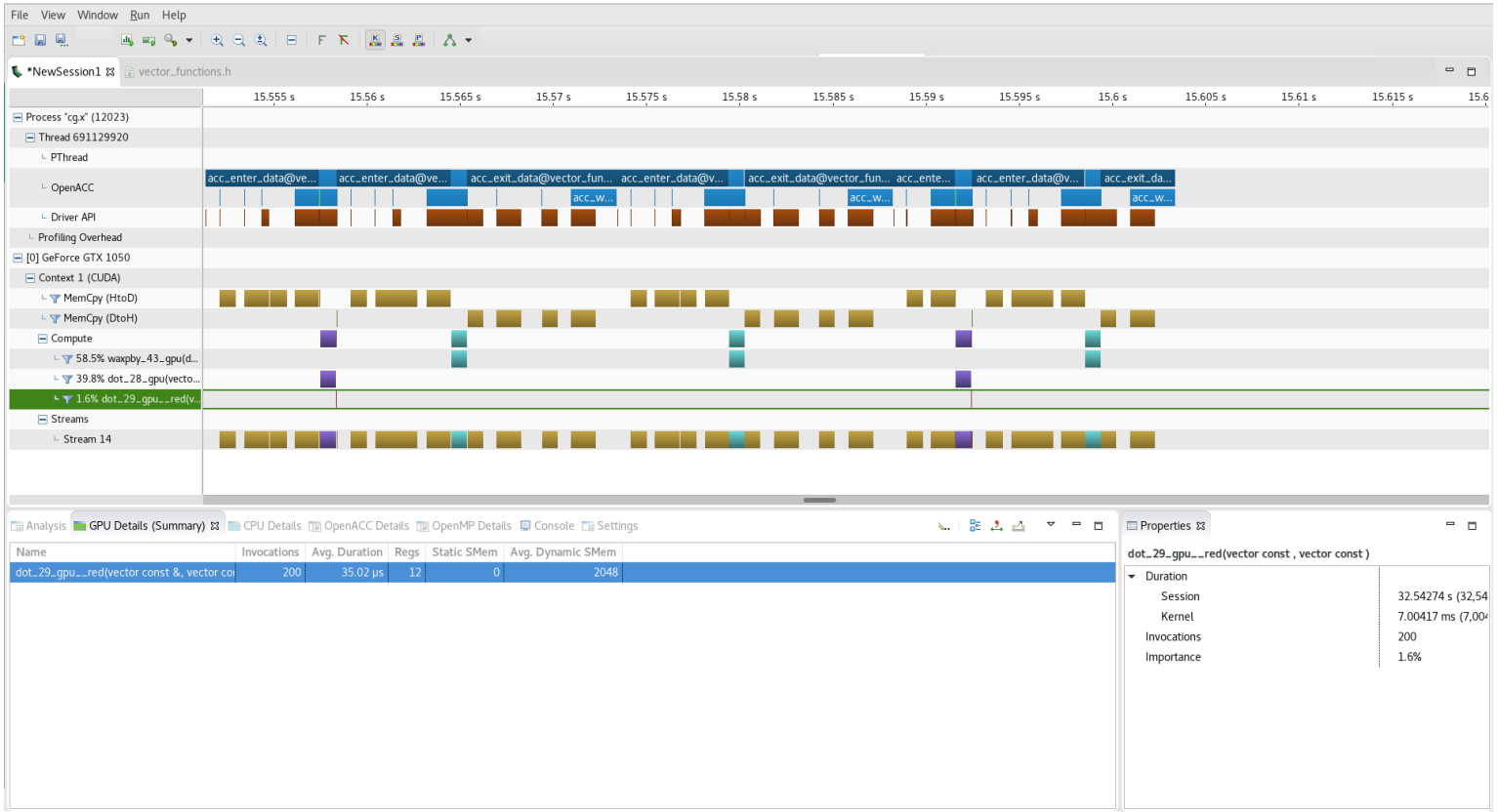
Accelerating DOT

Profile Timeline



Accelerating DOT

Profile Timeline



Accelerating MATVEC

First Step

- The first step is the addition of the OpenACC **kernel** pragma in the outermost nested loop.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels
30    for(int i=0;i<num_rows;i++) {
31      double sum=0;
32      int row_start=row_offsets[i];
33      int row_end=row_offsets[i+1];
34      for(int j=row_start;j<row_end;j++) {
35        unsigned int Acol=cols[j];
36        double Acoef=Acoefs[j];
37        double xcoef=xcoefs[Acol];
38        sum+=Acoef*xcoef;
39      }
40      ycoefs[i]=sum;
41    }
42  }
```

Accelerating MATVEC

First Step

- The compiler issues an error message e data dependency between interations of the loop;
- There is an accelerator restriction pointed out at line 34 of matrix_functions.h, which states that the compiler does not know how to calculate the size of the arrays used inside the kernel region, specifically the three arrays used in the innermost loop;
- The problem occurs because the loop bounds are hidden within the matrix data structure and the compiler cannot determine how to relocate the key arrays.

```
16 PGCC-S-0155-Compiler failed to translate accelerator region (see -Minfo messages): Could not find allocated-variable index for symbol- cols (main.cpp: 27)
17 matvec(const matrix &, const vector &, const vector &):
18       23, include "matrix_functions.h"
19           30, Loop is parallelizable
20               Generating Tesla code
21               30, #pragma acc loop gang /* blockIdx.x */
22               34, #pragma acc loop vector(128) /* threadIdx.x */
23               38, Generating implicit reduction(+:sum)
24           34, Accelerator restriction: size of the GPU copy of cols,xcoefs,Acoefs is unknown
25               Loop is parallelizable
26 PGCC-F-0704-Compilation aborted due to previous errors. (main.cpp)
27 PGCC/x86-64 Linux 19.10-0: compilation aborted
28 make: *** [cg.x] Error 2
```

Accelerating MATVEC

Second Step

- It is necessary to tell the compiler the shape of at least the three arrays used within the inner loop;
- One can use the copyin data clause since all three of these arrays are used only for reading.

```
20 void matvec(const matrix& A, const vector& x, const vector &y) {
21
22   unsigned int num_rows=A.num_rows;
23   unsigned int *restrict row_offsets=A.row_offsets;
24   unsigned int *restrict cols=A.cols;
25   double *restrict Acoefs=A.coefs;
26   double *restrict xcoefs=x.coefs;
27   double *restrict ycoefs=y.coefs;
28
29 #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30   for(int i=0;i<num_rows;i++) {
31     double sum=0;
32     int row_start=row_offsets[i];
33     int row_end=row_offsets[i+1];
34     for(int j=row_start;j<row_end;j++) {
35       unsigned int Acol=cols[j];
36       double Acoef=Acoefs[j];
37       double xcoef=xcoefs[Acol];
38       sum+=Acoef*xcoef;
39     }
40     ycoefs[i]=sum;
41   }
42 }
```

Accelerating MATVEC

Second Step

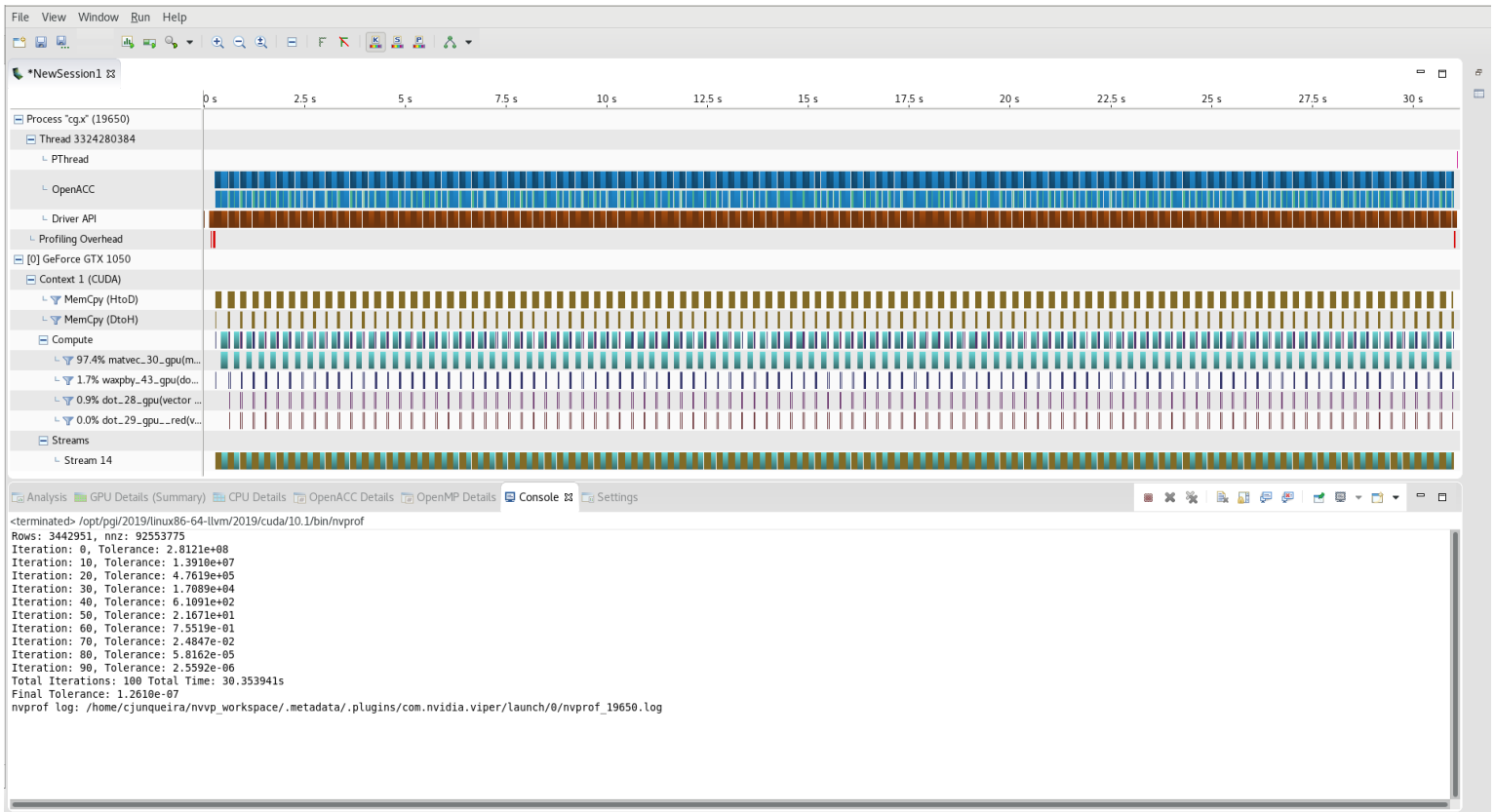
- The addition of a copy data clause to the kernels direction informs the compiler both how to allocate space of the affected variables on the accelerator, but also that data is only needed for as input to the loops.

```
16  matvec(const matrix &, const vector &, const vector &):
17       23, include "matrix_functions.h"
18             27, Generating copyin(Acoefs[:A->nnz]) [if not already present]
19                 Generating implicit copyin(row_offsets[:num_rows+1]) [if not already present]
20                 Generating copyin(cols[:A->nnz]) [if not already present]
21                 Generating implicit copyout(ycoefs[:num_rows]) [if not already present]
22                 Generating copyin(xcoefs[:x->n]) [if not already present]
23             30, Loop is parallelizable
24                 Generating Tesla code
25                 30, #pragma acc loop gang /* blockIdx.x */
26                 34, #pragma acc loop vector(128) /* threadIdx.x */
27                 38, Generating implicit reduction(+:sum)
28             34, Loop is parallelizable
```

Accelerating MATVEC

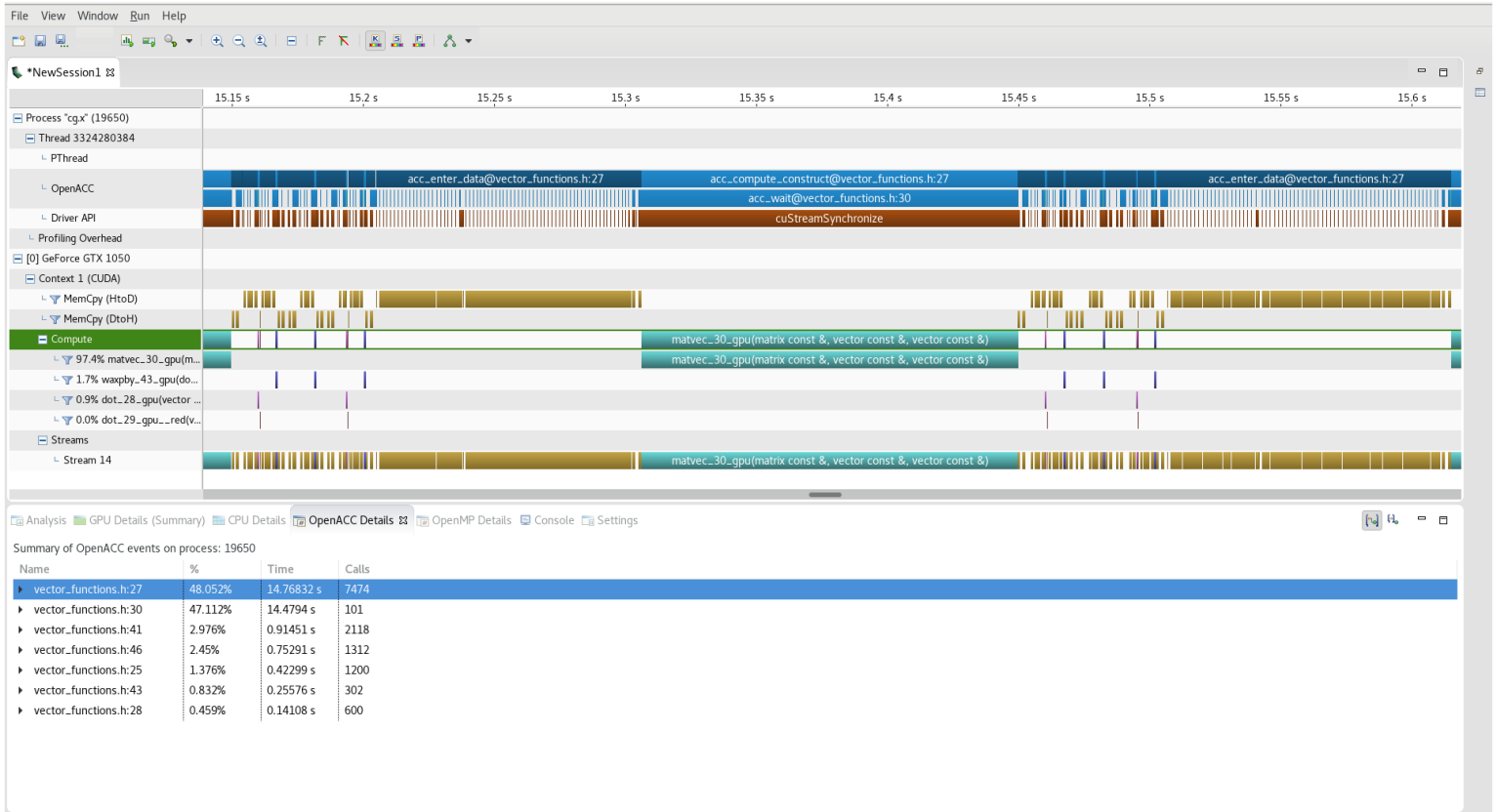
Second Step

- The runtime after the modifications is longer than the original code;
- There are a lot of data transfers along the calculations;



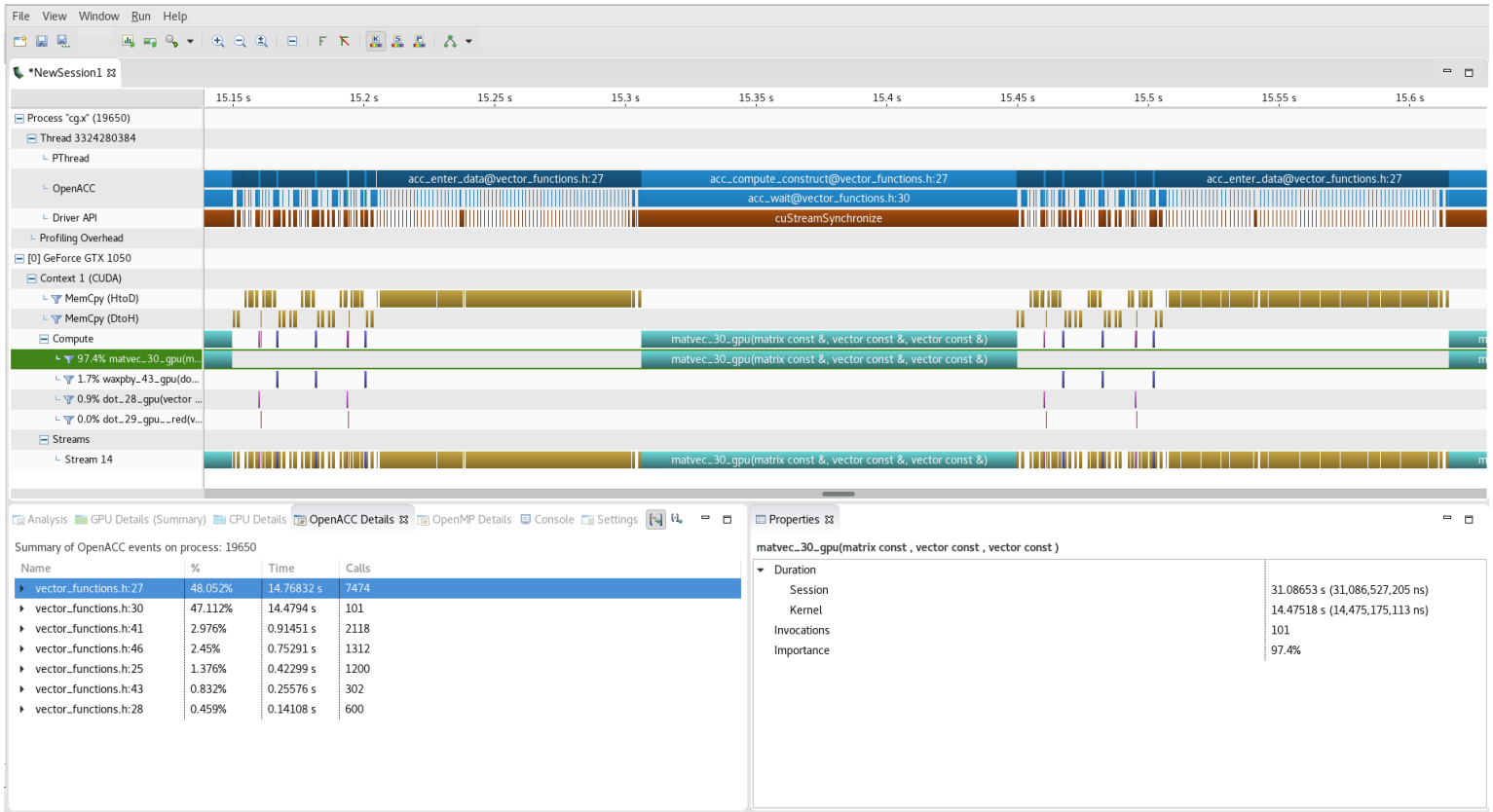
Accelerating MATVEC

Second Step



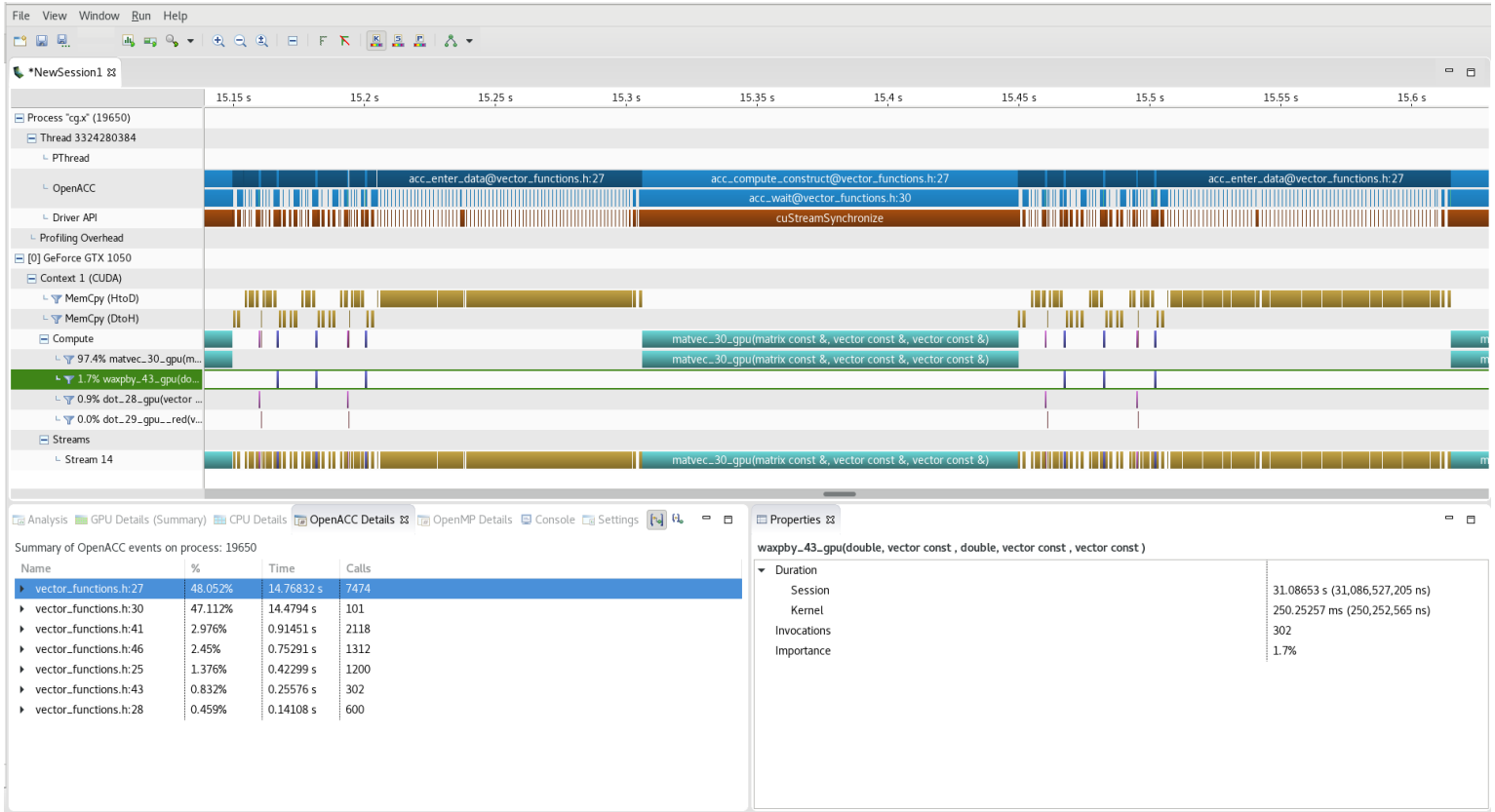
Accelerating MATVEC

Second Step



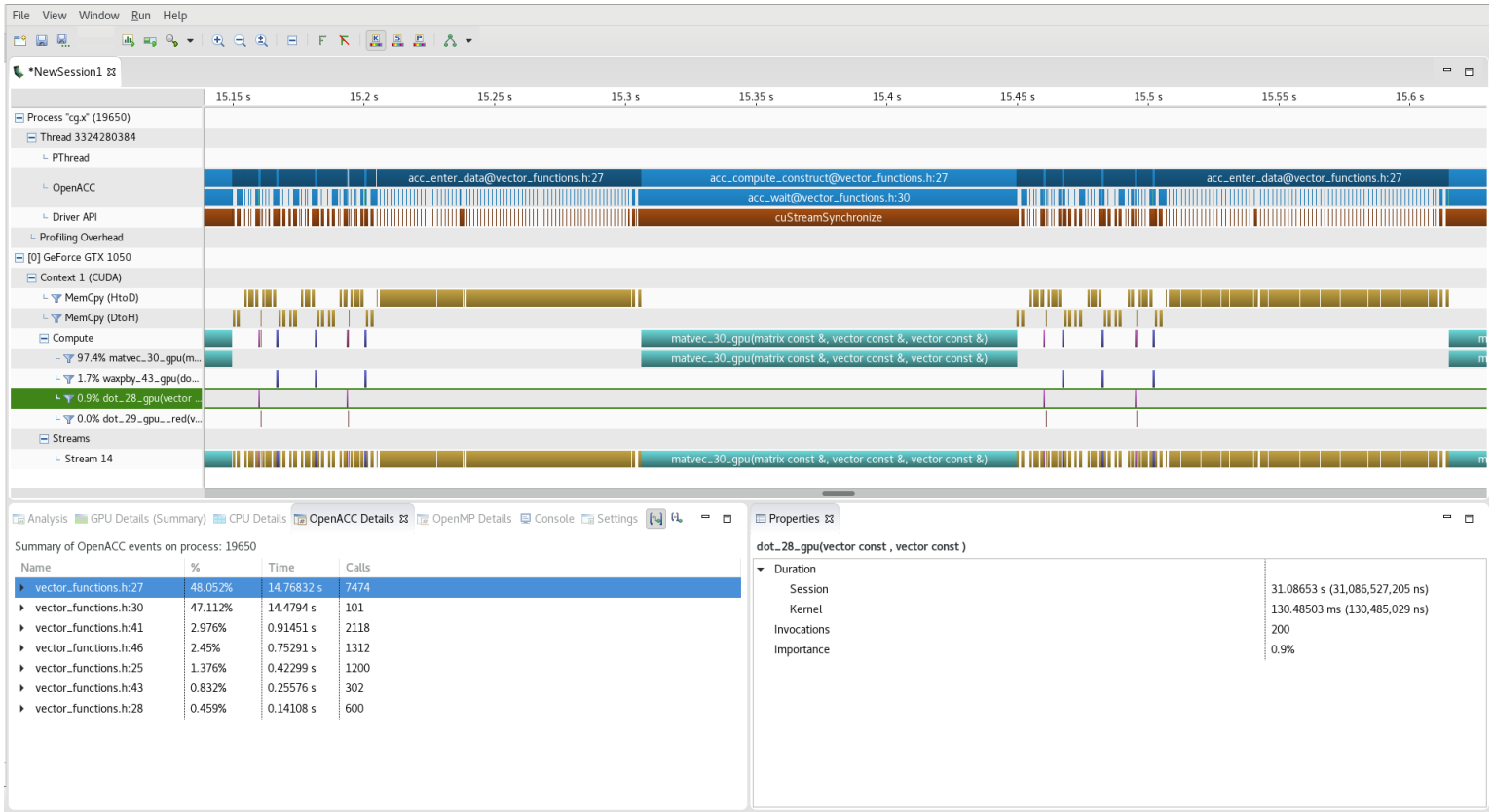
Accelerating MATVEC

Second Step



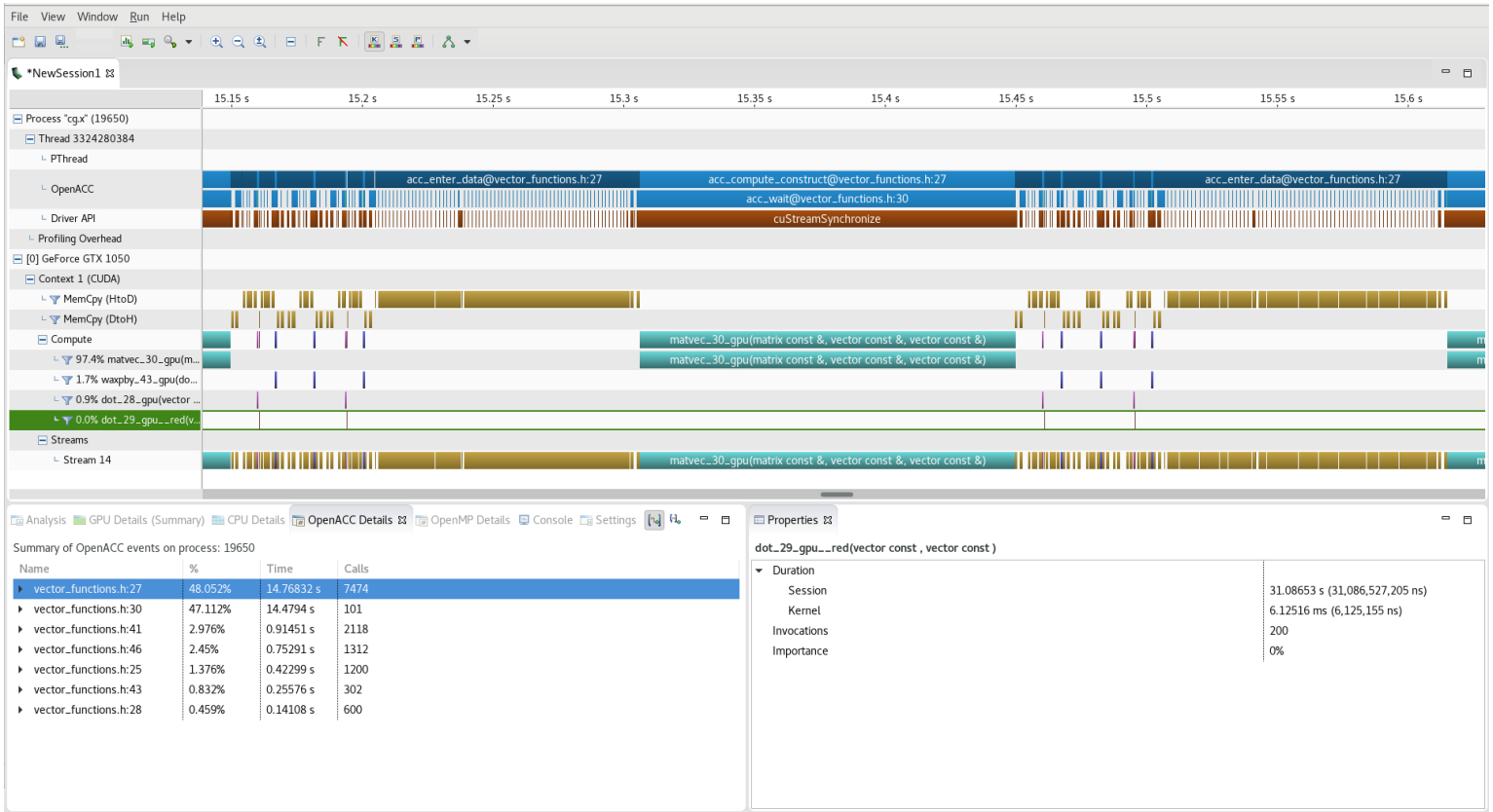
Accelerating MATVEC

Second Step



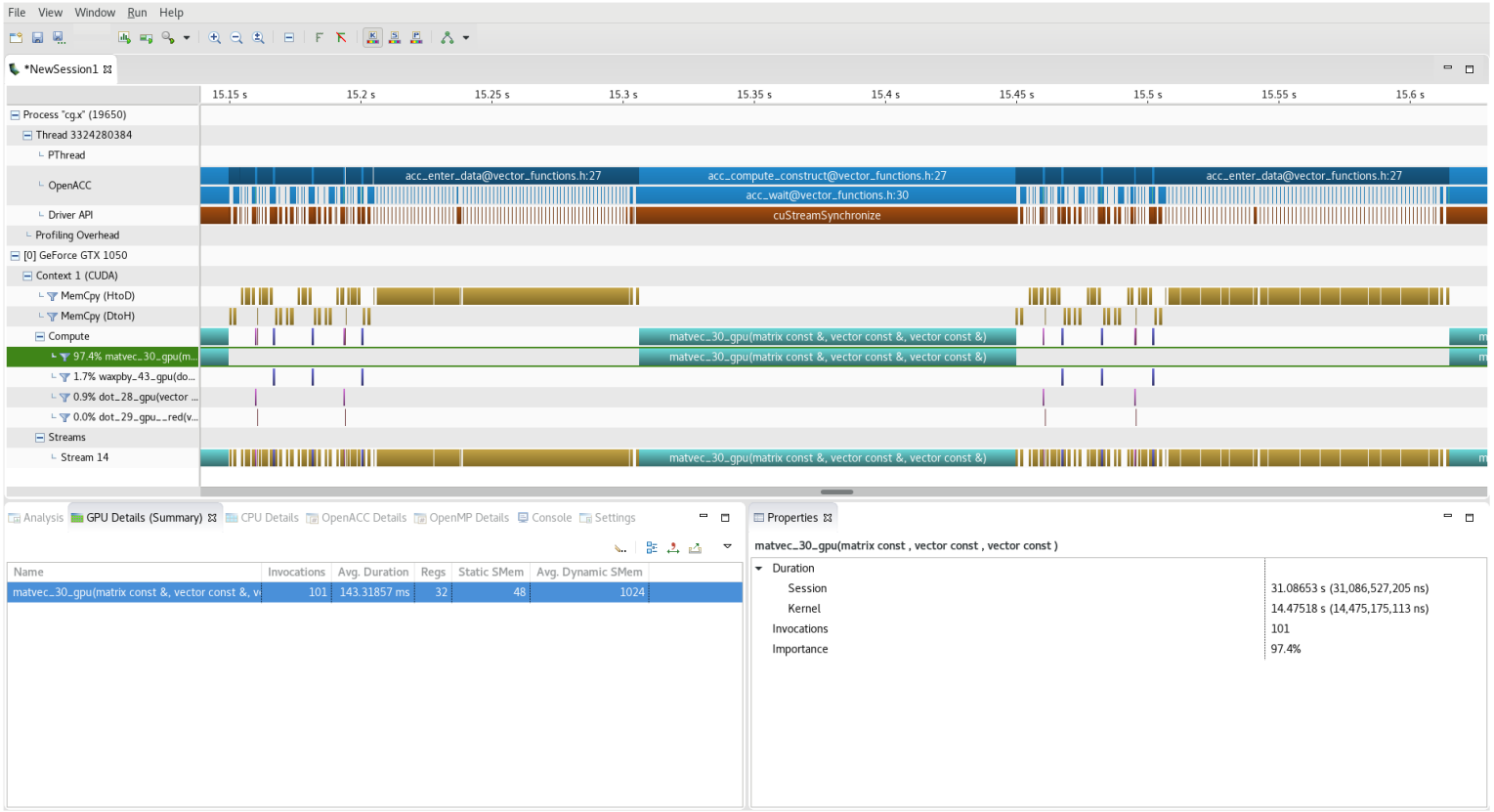
Accelerating MATVEC

Second Step



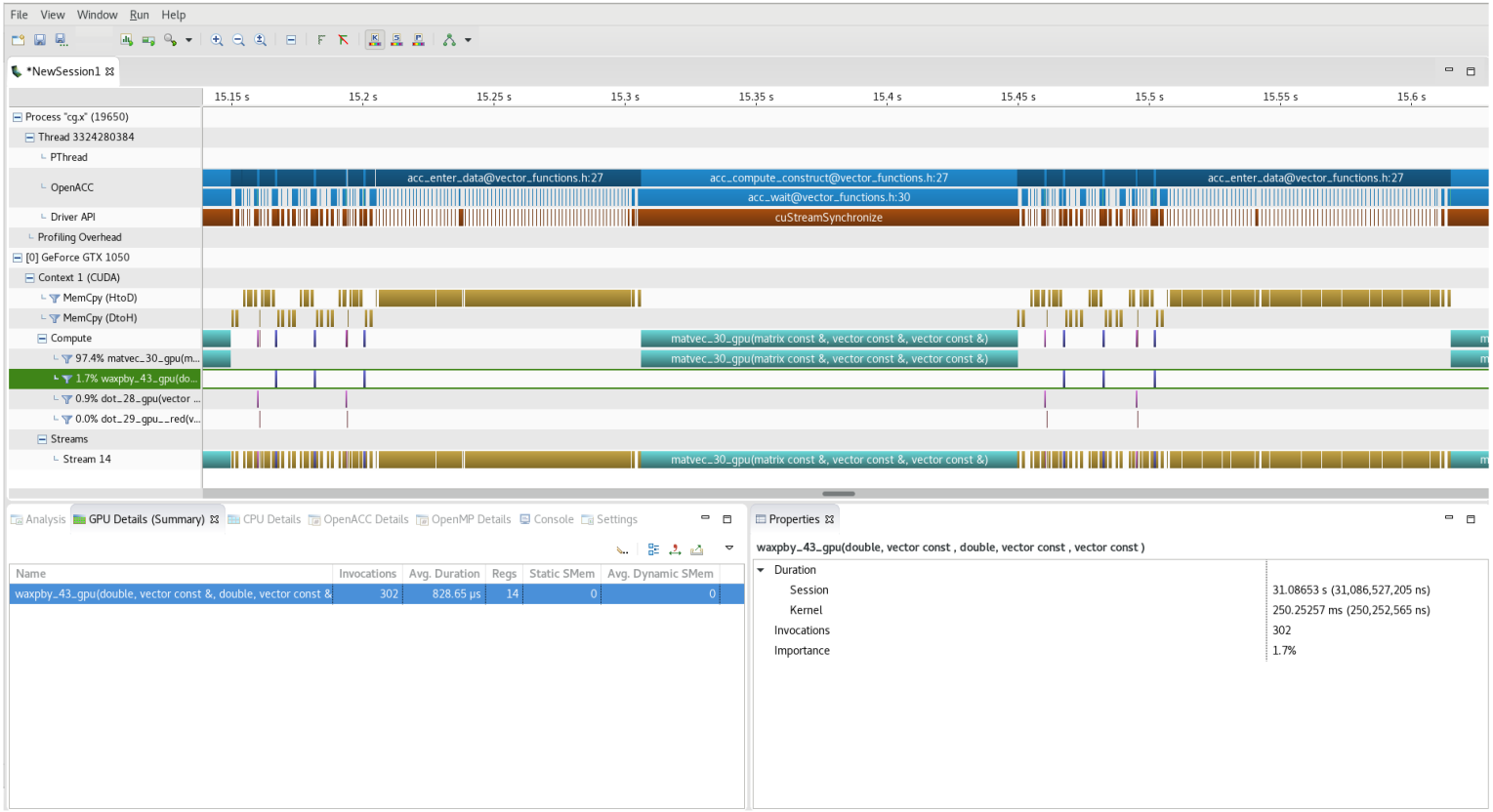
Accelerating MATVEC

Second Step



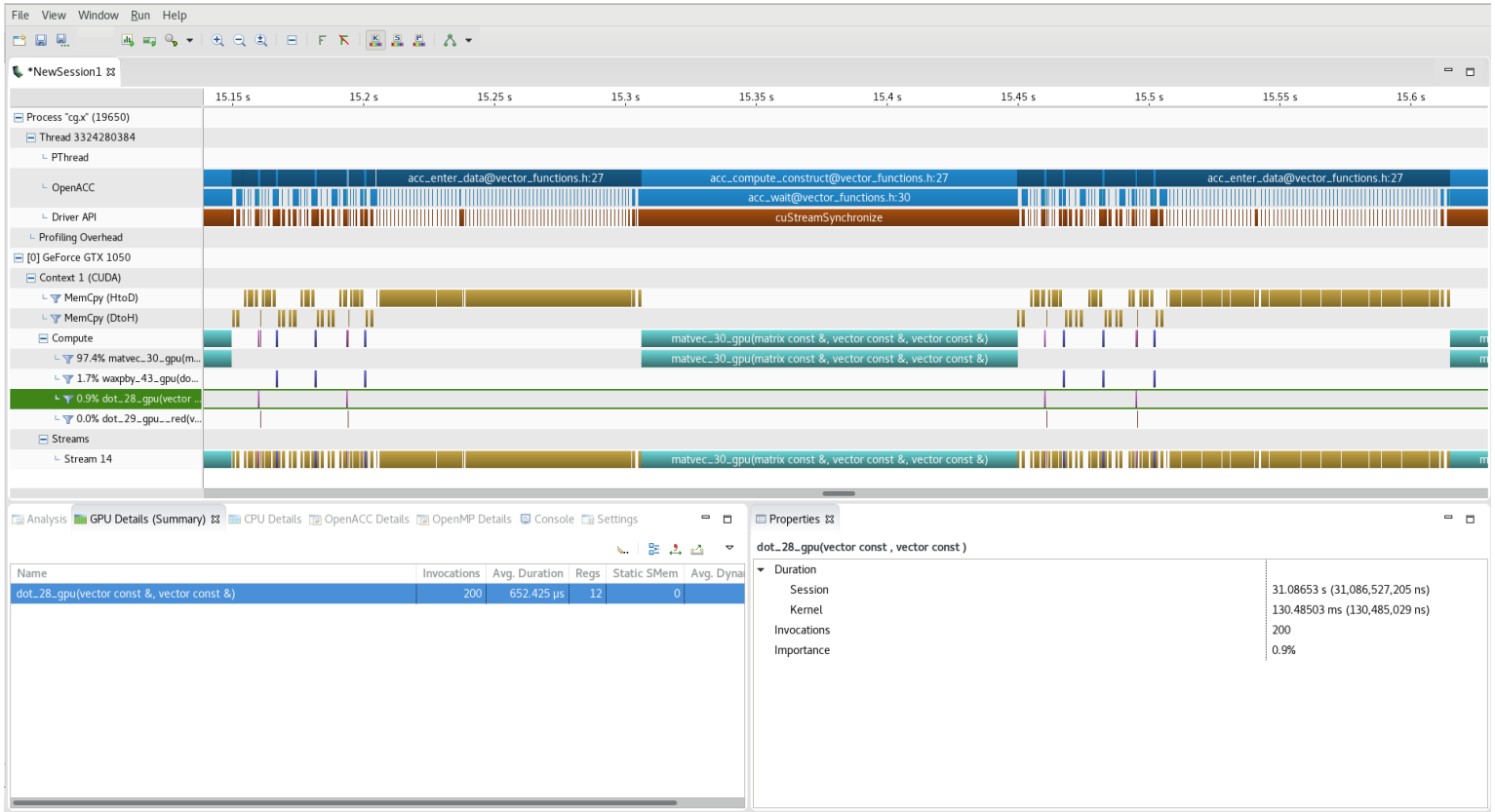
Accelerating MATVEC

Second Step



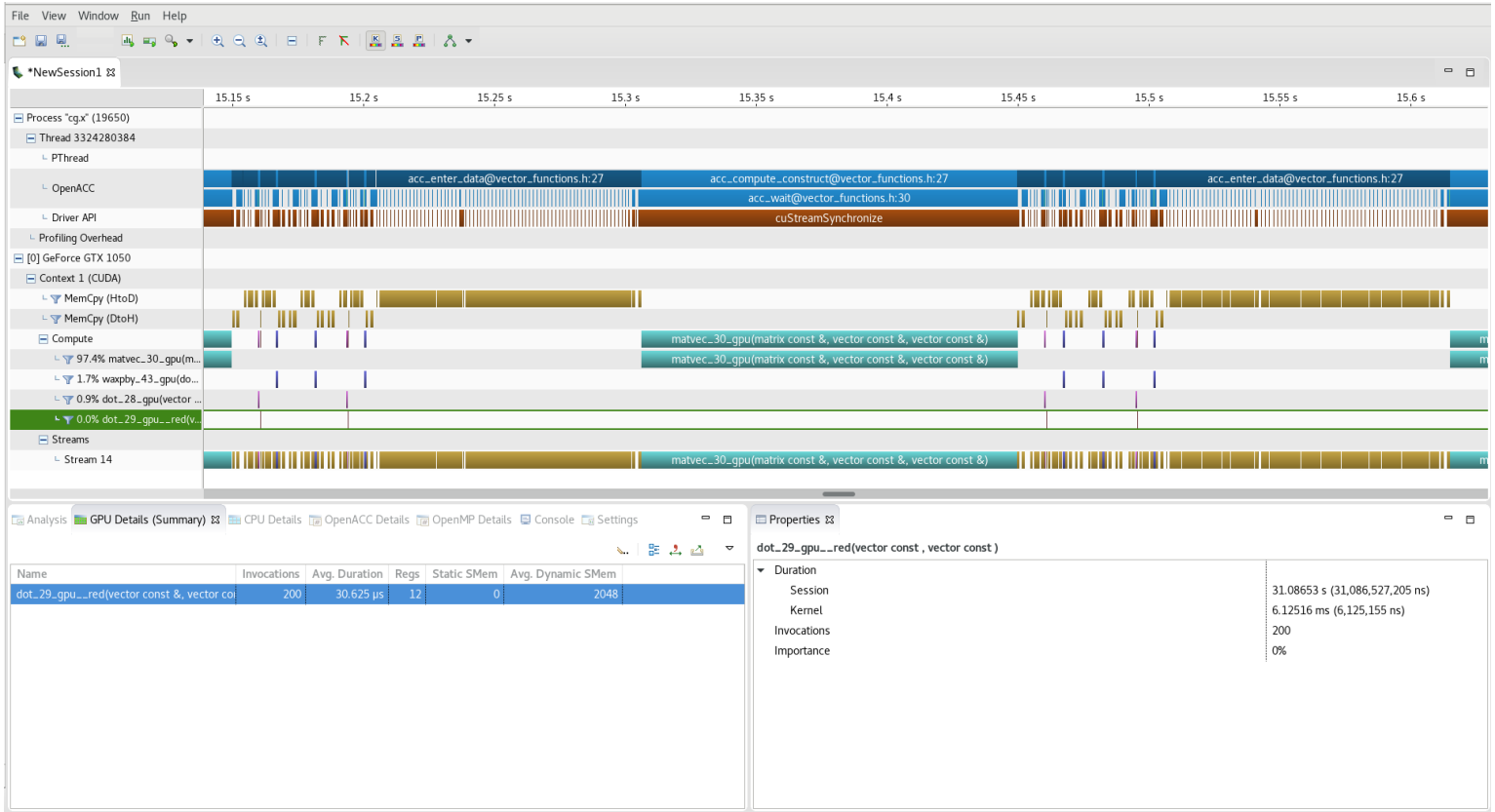
Accelerating MATVEC

Second Step



Accelerating MATVEC

Second Step



Data Movement

Matrix

- OpenACC data model is host-first, meaning that data directives that allocate data on the device must appear after host allocations and data directives that deallocate device data must appear before host deallocations;
- The matrix allocation is performed in **allocate_3d_poisson_matrix** function;
- The function does perform an initialization of the matrix, Therefore, one can add copy clauses at the end of the routine so that the device copy of the data structure will contain initialized data;

```
69      A.row_offsets[num_rows]=nnz;
70      A.nnz=nnz;
71    #pragma acc enter data copyin(A)
72    #pragma acc enter data copyin(A.row_offsets[0:num_rows+1],A.cols[0:nnz],A.coefs[0:nnz])
73    }
74
75    void free_matrix(matrix &A) {
76      unsigned int *row_offsets=A.row_offsets;
77      unsigned int * cols=A.cols;
78      double * coefs=A.coefs;
79
80    #pragma acc exit data delete(A.row_offsets,A.cols,A.coefs)
81    #pragma acc exit data delete(A)
82      free(row_offsets);
83      free(cols);
84      free(coefs);
85    }
```

Data Movement

Matrix

- The A structure, used in **allocate_3d_possion_matrix** function, contains two scalar numbers, along with 3 pointers that are used on the device;
- The copyin of *A* at line 71 copies the structure, with the 5 elements that were described here. When the 3 pointers are copied, however, they will contain pointers back to the host memory, since the pointers were simply copied on the device;
- It is then necessary to copy the data pointed to by those pointers, which is done in line 72. it gives the three arrays contained in A shapes and copies their data to the device;
- When removing the data from the device in the **free_matrix** function, the data transferts should be done in the reverse order of what was performed in **allocate_3d_possion_matrix** function.

```
69    A.row_offsets[num_rows]=nnz;
70    A.nnz=nnz;
71  #pragma acc enter data copyin(A)
72  #pragma acc enter data copyin(A.row_offsets[0:num_rows+1],A.cols[0:nnz],A.coefs[0:nnz])
73  }
74
75  void free_matrix(matrix &A) {
76    unsigned int *row_offsets=A.row_offsets;
77    unsigned int * cols=A.cols;
78    double * coefs=A.coefs;
79
80  #pragma acc exit data delete(A.row_offsets,A.cols,A.coefs)
81  #pragma acc exit data delete(A)
82    free(row_offsets);
83    free(cols);
84    free(coefs);
85  }
```

Data Movement

Vector

- The **allocate_vector** and **free_vector** functions can be similarly modified to relocate Vector structures to the accelerator as well;
- Since **allocate_vector** function does not initialize the vector with any data, the **create** data clause can be used to allocate memory without causing any data copies to occur;
- An update directive can be used in **initialize_vector** function to copy the data in the initialized host vector to the device.

```
24 void allocate_vector(vector &v, unsigned int n) {
25   v.n=n;
26   v.coefs=(double*)malloc(n*sizeof(double));
27 #pragma acc enter data create(v)
28 #pragma acc enter data create(v.coefs[0:n])
29 }
30
31 void free_vector(vector &v) {
32 #pragma acc exit data delete(v.coefs)
33 #pragma acc exit data delete(v)
34   free(v.coefs);
35   v.n=0;
36 }
37
38 void initialize_vector(vector &v,double val) {
39
40   for(int i=0;i<v.n;i++)
41     v.coefs[i]=val;
42 #pragma acc update device(v.coefs[0:v.n])
43 }
```

Data Movement

Compiling

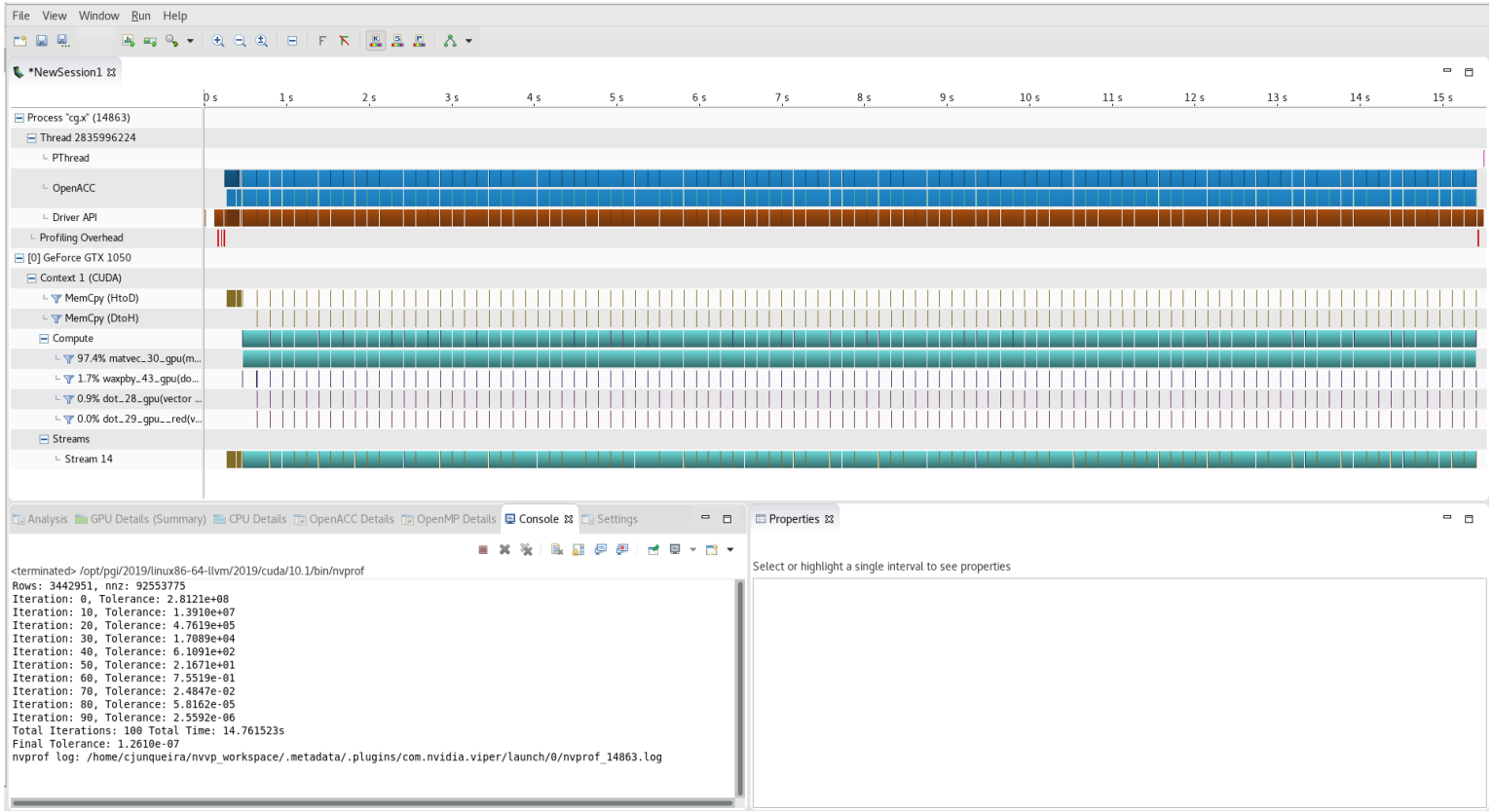
- The compiler issues the data movement implemented in the code.

```
3       20, include "vector.h"
4            29, Accelerator clause: upper bound for dimension 0 of array 'v' is unknown
5                Generating enter data create(v[:1],v->coefs[:n])
6   free_vector(vector &):
7       20, include "vector.h"
8            34, Generating exit data delete(v[:1],v->coefs[:1])
9   initialize_vector(vector &, double):
10      20, include "vector.h"
11           43, Generating update device(v->coefs[:v->n])
12  dot(const vector &, const vector &):
```

```
27      22, include "matrix.h"
28           73, Accelerator clause: upper bound for dimension 0 of array 'A' is unknown
29               Generating enter data copyin(A[:1],A->row_offsets[:num_rows+1],A->coefs[:nnz],A->cols[:nnz])
30  free_matrix(matrix &):
31      22, include "matrix.h"
32           82, Generating exit data delete(A->coefs[:1],A->cols[:1],A[:1],A->row_offsets[:1])
33  matvec(const matrix &, const vector &, const vector &):
```

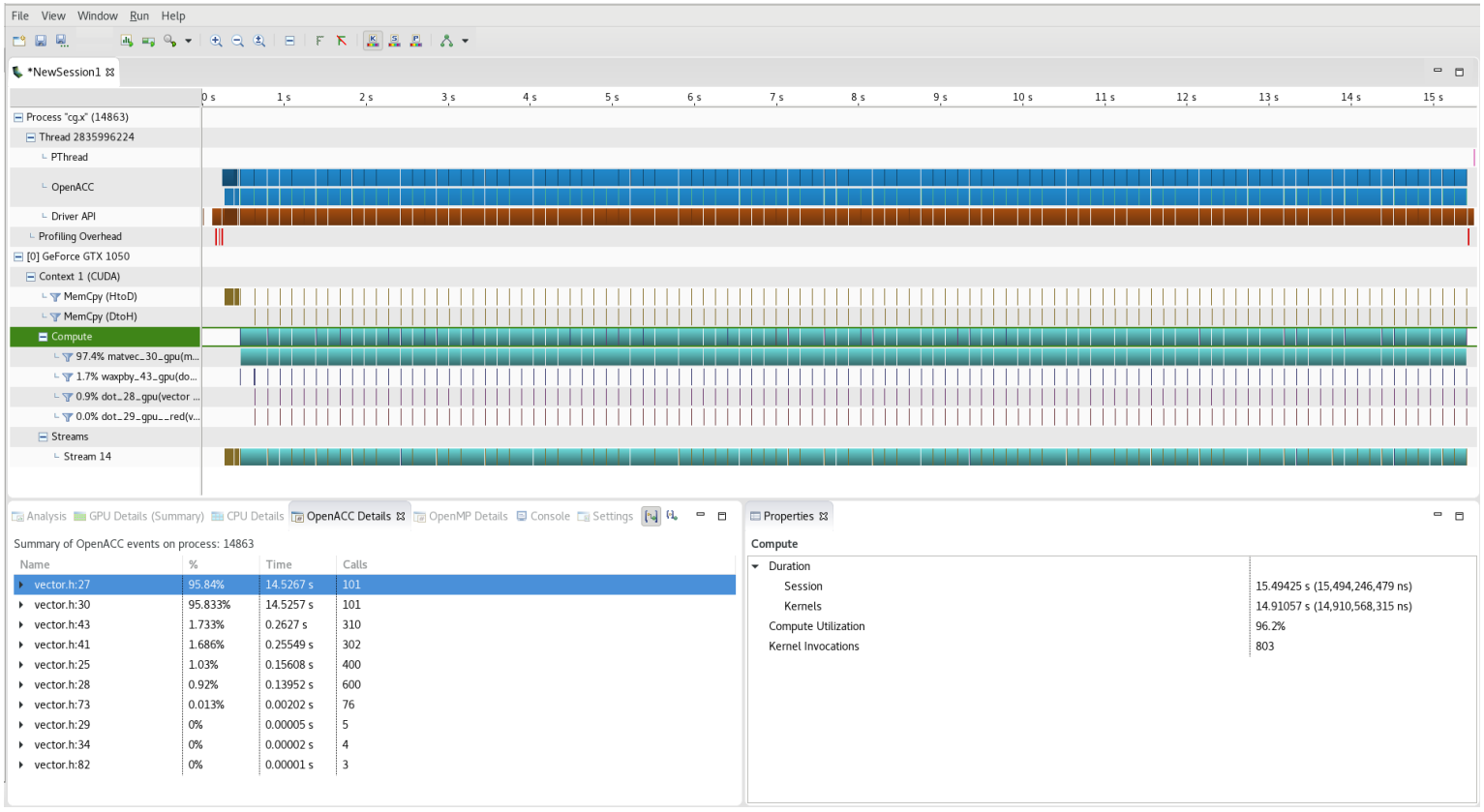
Data Movement

Profile Timeline



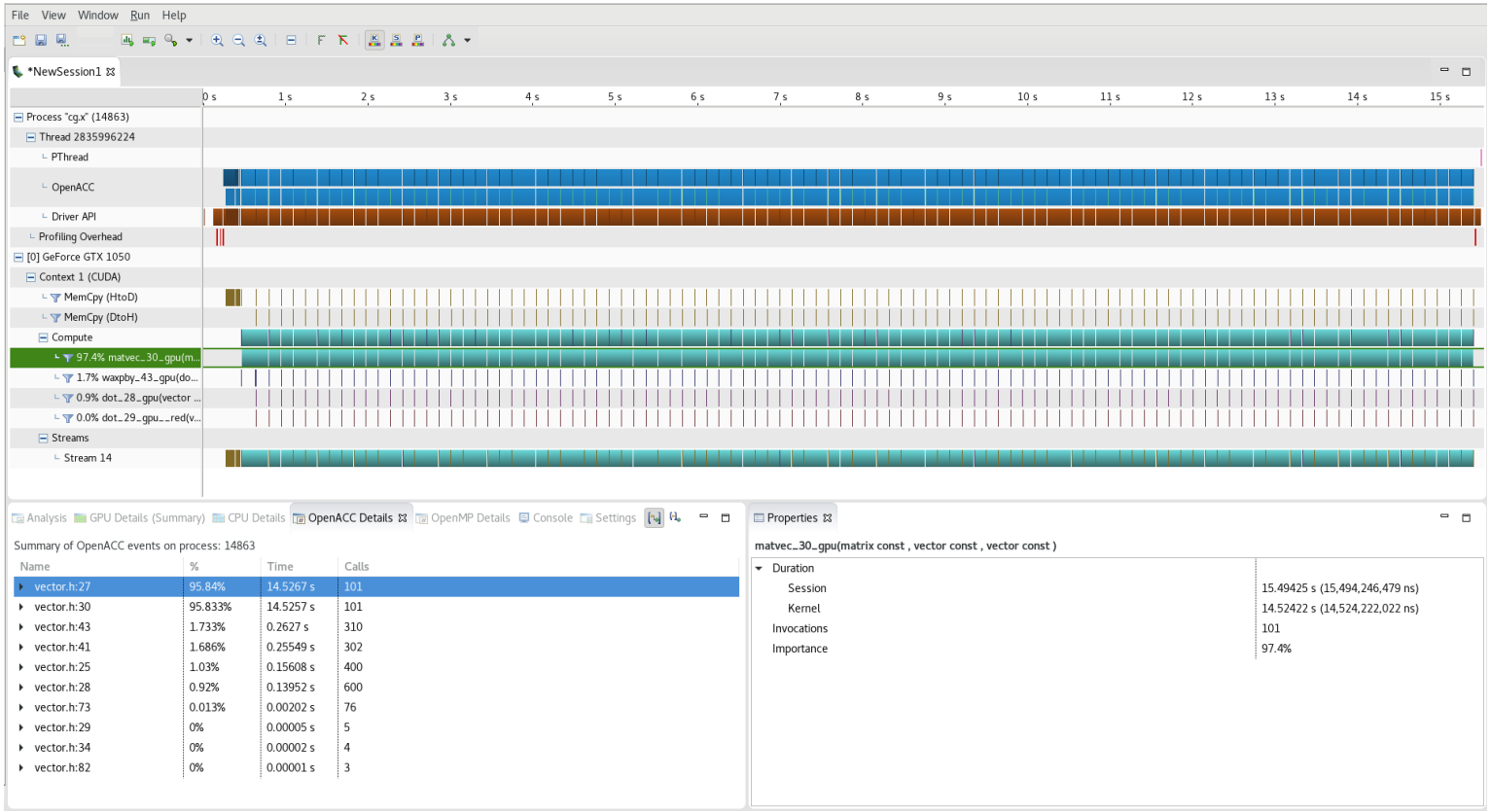
Data Movement

Profile Timeline



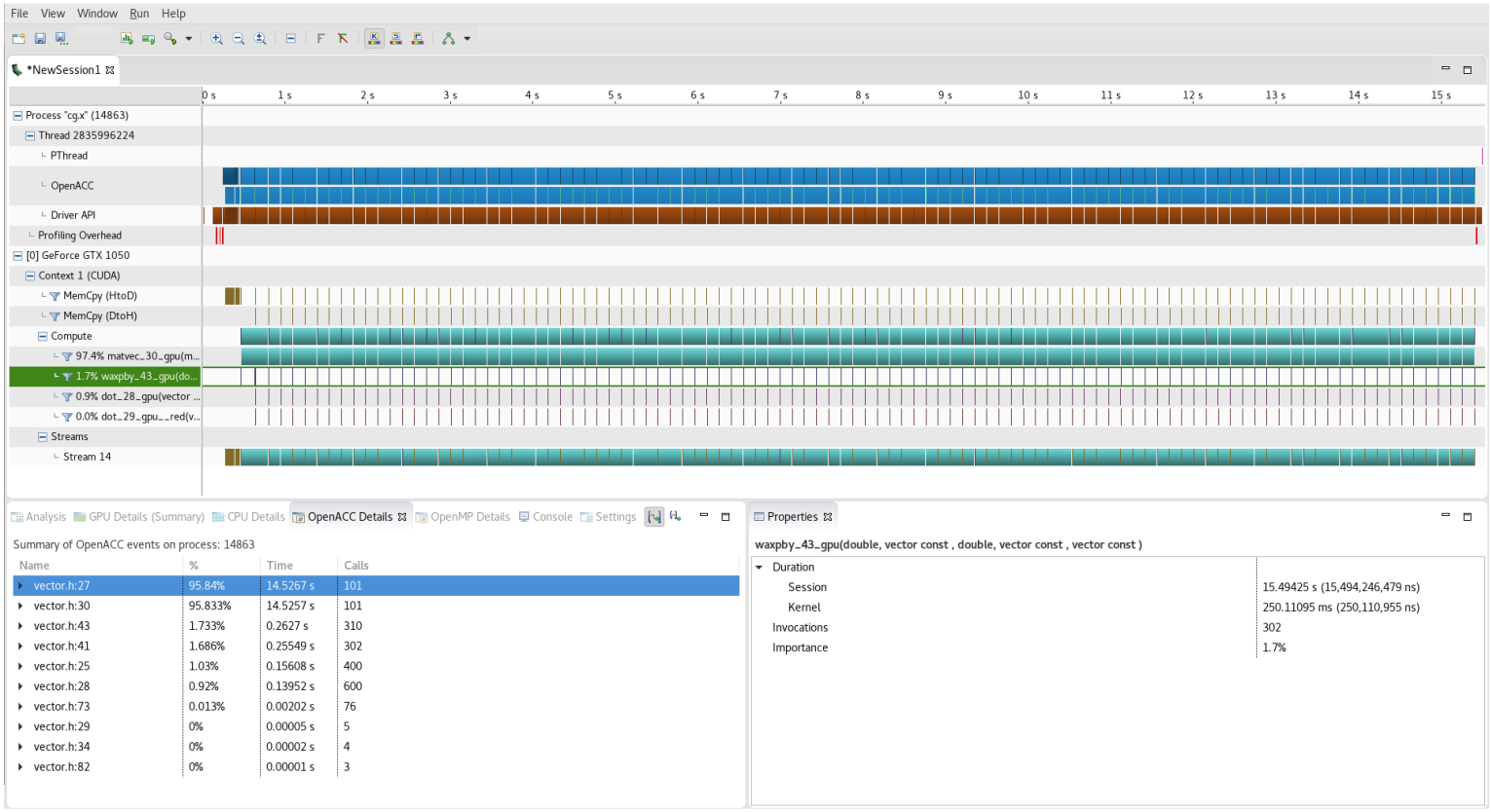
Data Movement

Profile Timeline



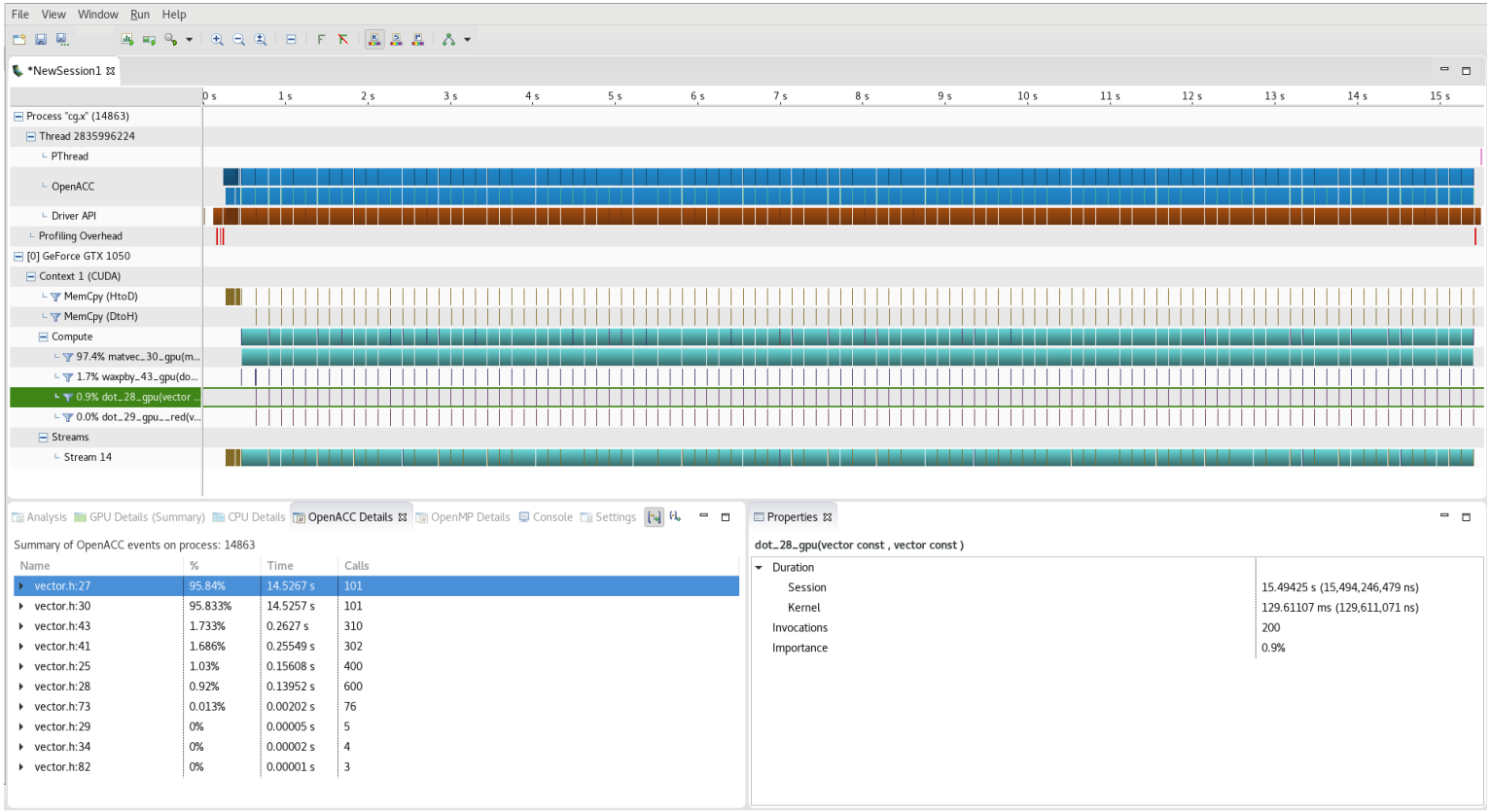
Data Movement

Profile Timeline



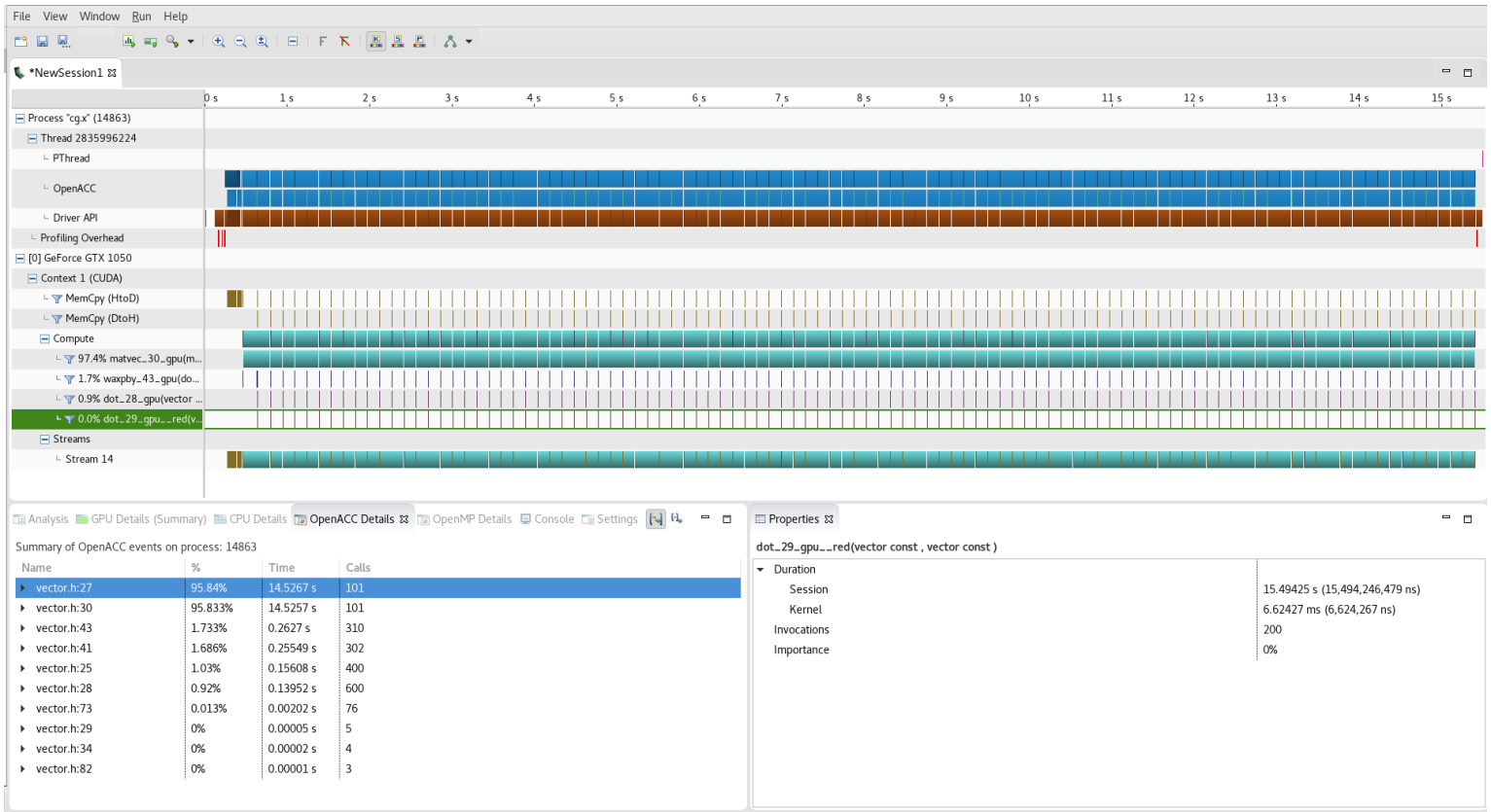
Data Movement

Profile Timeline



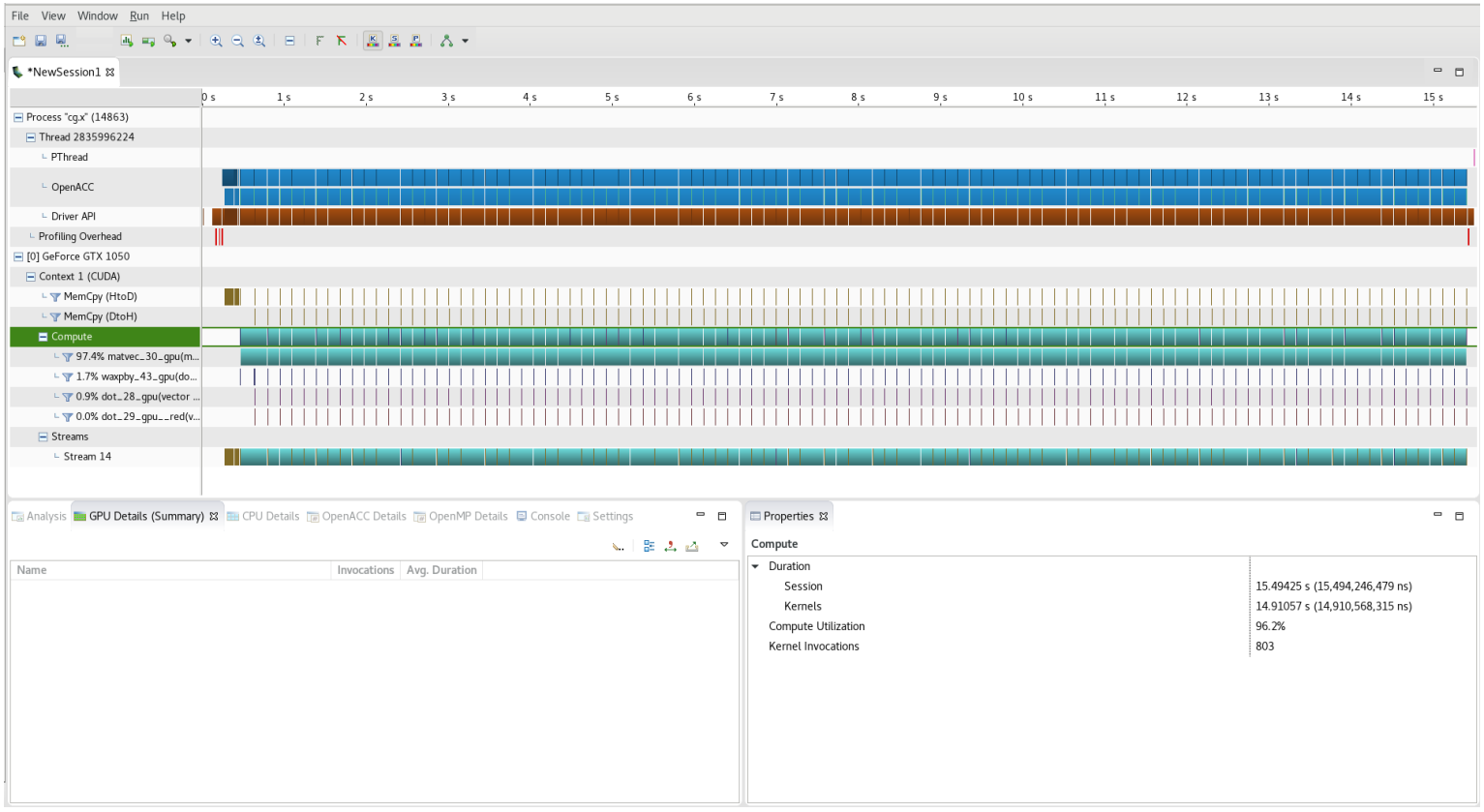
Data Movement

Profile Timeline



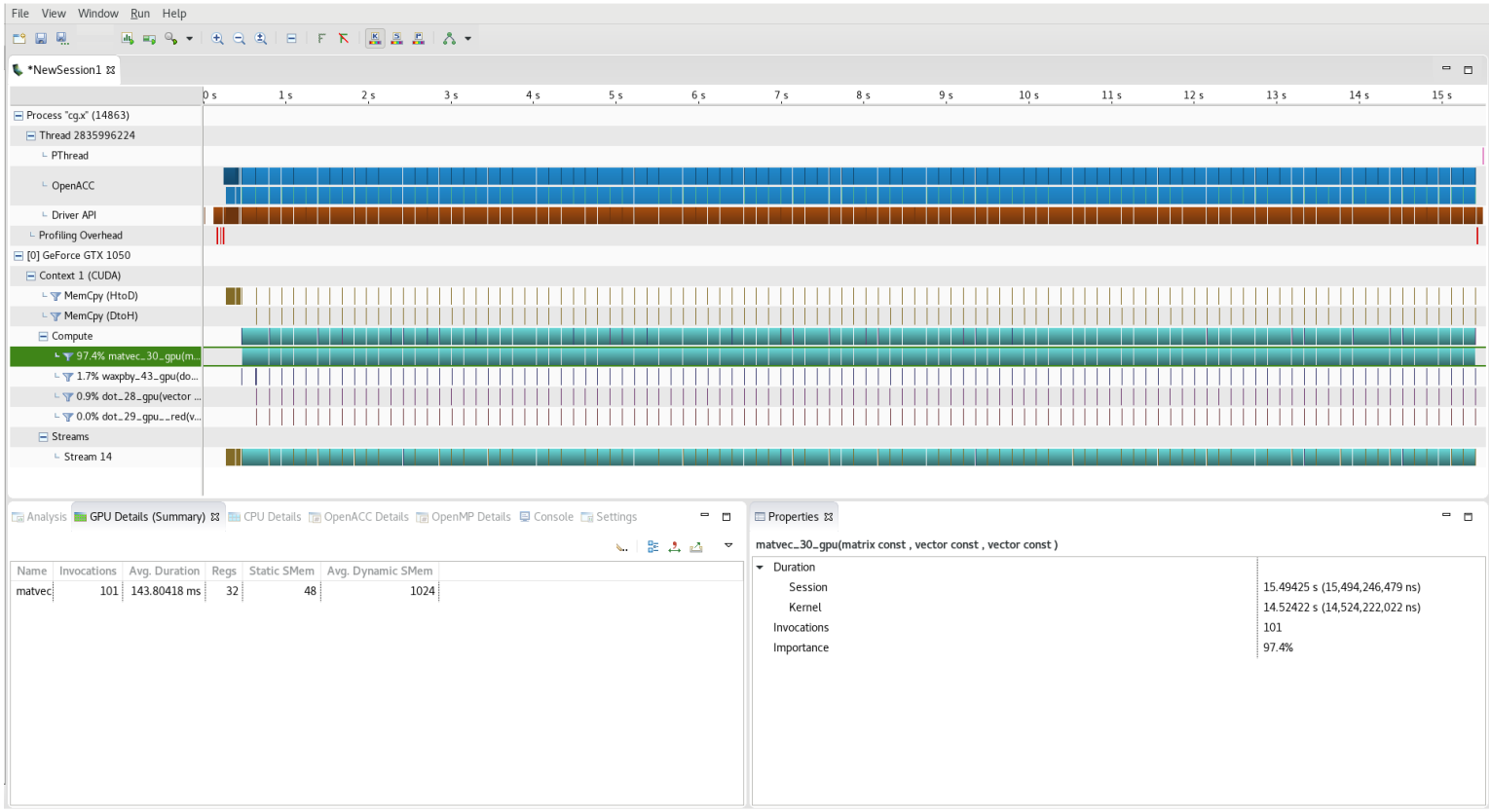
Data Movement

Profile Timeline



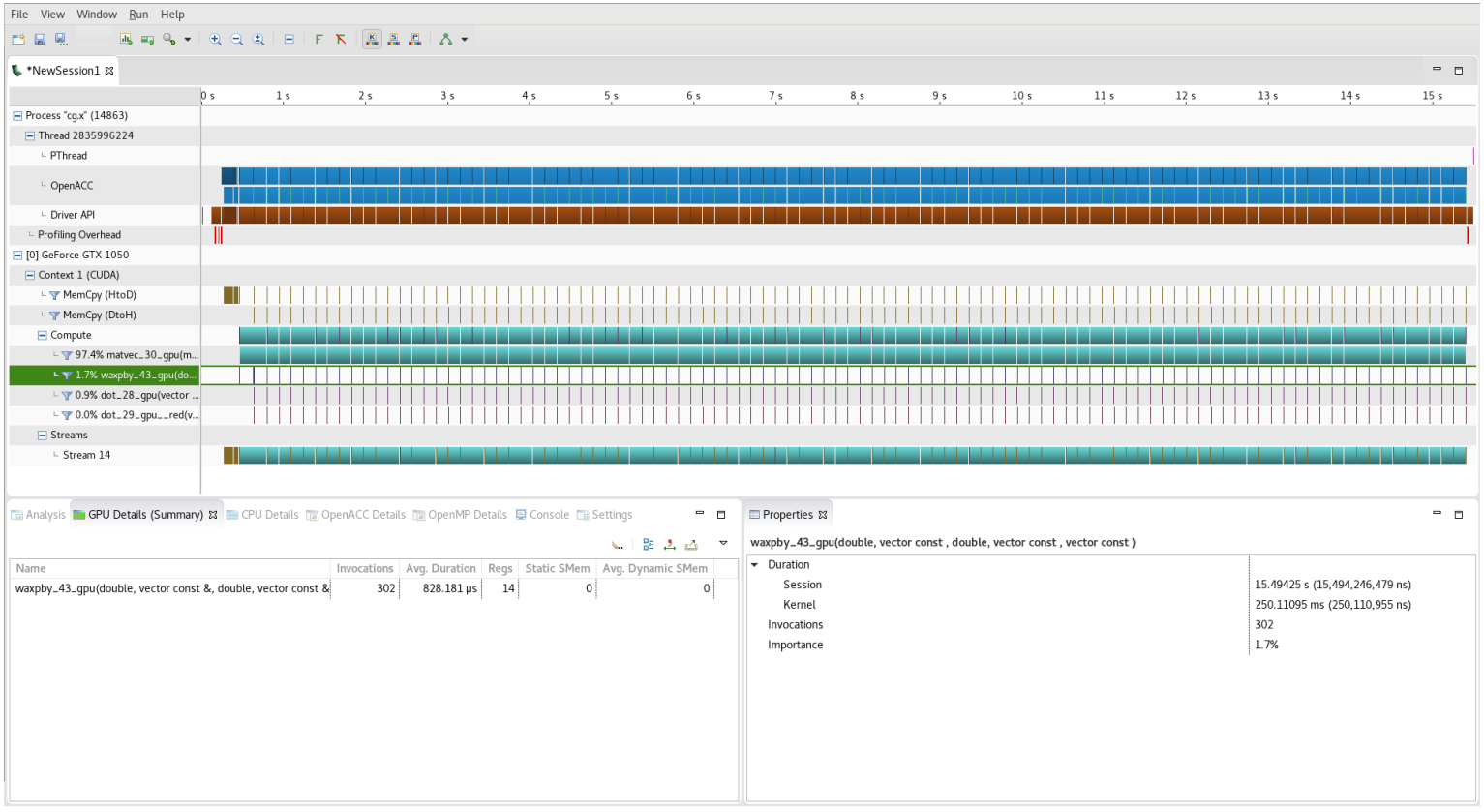
Data Movement

Profile Timeline



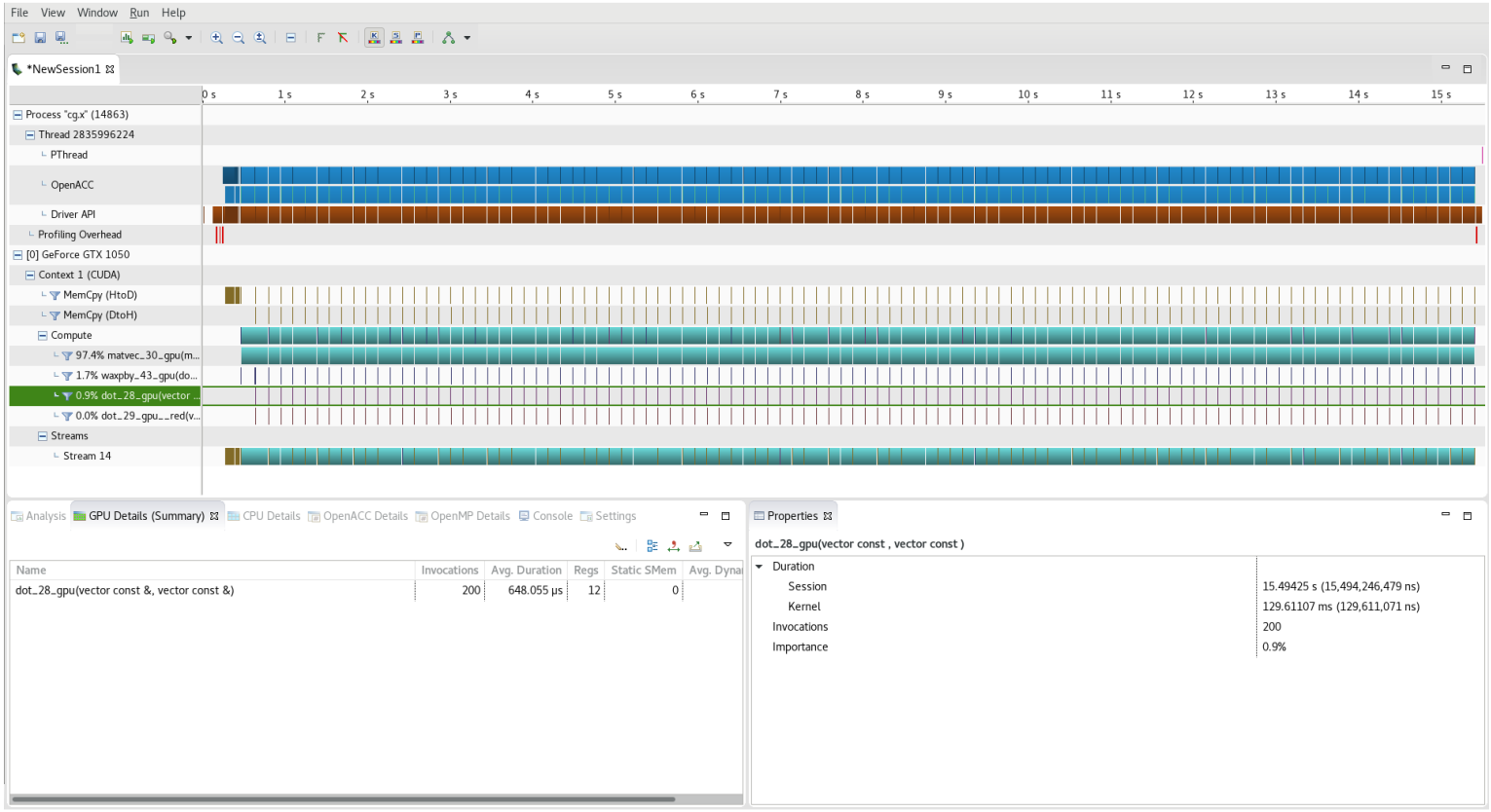
Data Movement

Profile Timeline



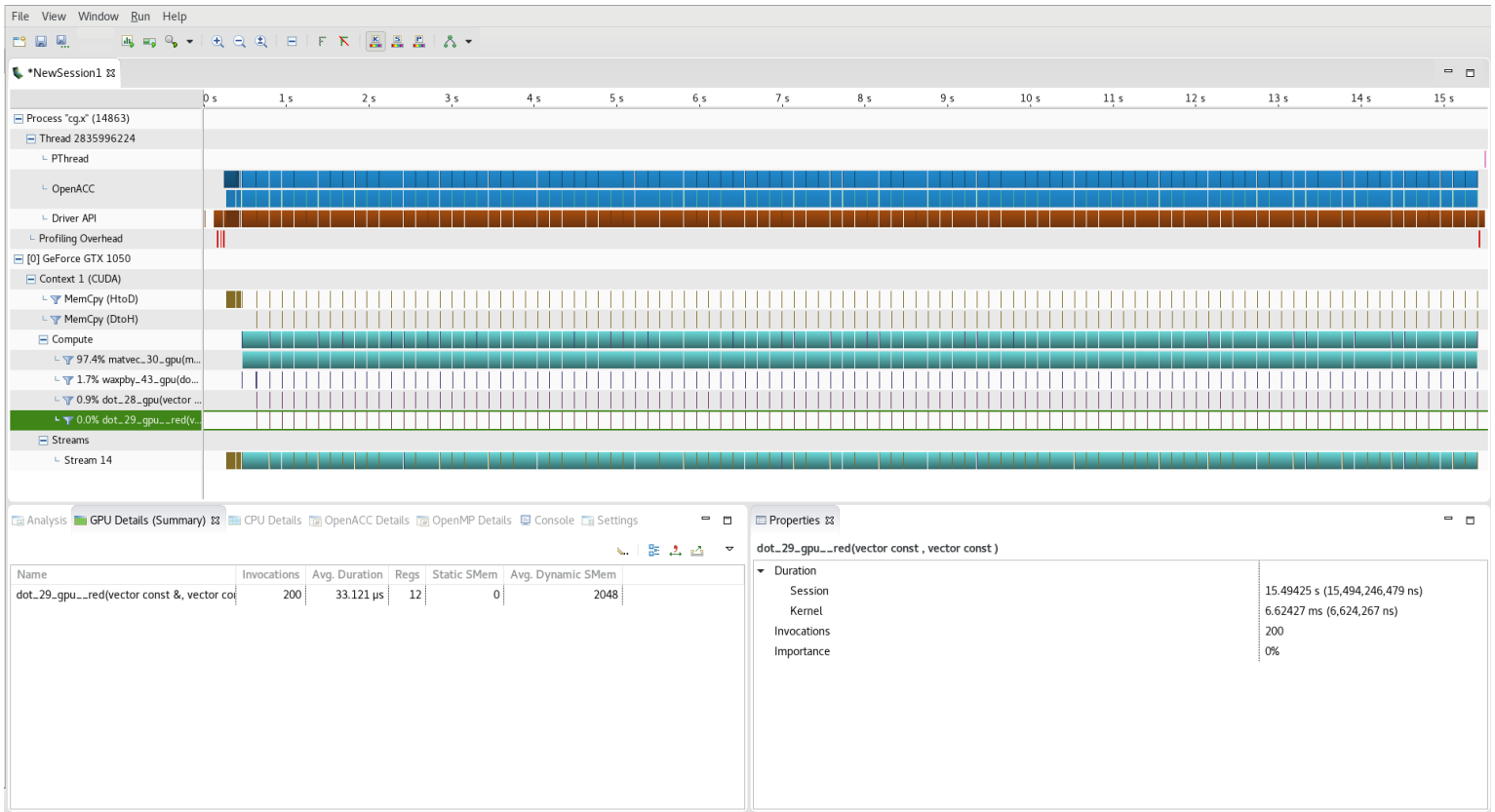
Data Movement

Profile Timeline



Data Movement

Profile Timeline



Optimize Loops

Target-specific Optimizations

- The code is now running 2x faster than the original serial code;
- One can further improve the performance when applying target-specific optimizations;
- OpenACC provides a means for specifying a **devide_type** for optimizations so that the additional clauses will only apply when built for the specified device type;

Optimize Loops

A Review on OpenACCs Levels of Parallelis

- *Vector* parallelism is very tight-grained parallelism where multiple data elements are operated on with the same instruction. For instance, if there are two arrays of number that need to be added together, some hardware has special instructions that are able to operate on groups of numbers from each array and add them all together at the same time;
- The *vector length* is referred as how many number get added together by the same instructions and it depends on the architecture of the system;
- *Gangs* are at the top of the levels of parallelism and they can operate completely independently of each other, with no synchronization and no guarantees about when each gang will run in relation to other gangs. Gang level is very scalable because its very coarse-grained parallelism;
- *Worker* parallelism falls in between gang and vector. A gang is comprised of one or more workers, each of which operate on a vector;
- The workers within a vector have the ability to synchronize and may share a common cache of very fast memory;
- Every loop parallelized by an OpenACC compiler will be mapped to at least one of the three levels of parallelism, or it will be sequentially, *a.k.a "seq"*.

Optimize Loops

Vector Parallelism

- Compiler feedback from the version of the code with data movement optimizations indicates a gang level of parallelism for the outermost loop in line 30 and a vector level of parallelism, with a vector length of 128, for the innermost loop in line 34;

```
34      23, include "matrix_functions.h"
35          27, Generating copyin(Acoefs[:A->nnz]) [if not already present]
36              Generating implicit copyin(row_offsets[:num_rows+1]) [if not already present]
37              Generating copyin(cols[:A->nnz]) [if not already present]
38              Generating implicit copyout(ycoefs[:num_rows]) [if not already present]
39              Generating copyin(xcoefs[:x->n]) [if not already present]
40          30, Loop is parallelizable
41              Generating Tesla code
42              30, #pragma acc loop gang /* blockIdx.x */
43              34, #pragma acc loop vector(128) /* threadIdx.x */
44              38, Generating implicit reduction(+:sum)
45          34, Loop is parallelizable
```

Optimize Loops

Vector Parallelism

- NVIDIA GPUs perform best when the vector loop access data in a *stride-1* fashion, meaning each successive loop iteration accesses a successive array element;
- Although the outer loop accesses the *row_offsets* array *stride-1*, the inner loop accesses the *Acol*, *Acoefs*, and *xcoefs* arrays in *stride-1* manner, making it a better target for vectorization;
- One can add a **loop** pragma above the inner loop to specify the vector length.

matrix_functions.h

```
20 void matvec(const matrix& A, const vector& x, const vector &y) {
21
22   unsigned int num_rows=A.num_rows;
23   unsigned int *restrict row_offsets=A.row_offsets;
24   unsigned int *restrict cols=A.cols;
25   double *restrict Acoefs=A.coefs;
26   double *restrict xcoefs=x.coefs;
27   double *restrict ycoefs=y.coefs;
28
29 #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30   for(int i=0;i<num_rows;i++) {
31     double sum=0;
32     int row_start=row_offsets[i];
33     int row_end=row_offsets[i+1];
34     for(int j=row_start;j<row_end;j++) {
35       unsigned int Acol=cols[j];
36       double Acoef=Acoefs[j];
37       double xcoef=xcoefs[Acol];
38       sum+=Acoef*xcoef;
39     }
40     ycoefs[i]=sum;
41   }
42 }
```

Optimize Loops

Reduce Vector Length

- The inner loop always operates on 27 non-zero elements. Therefore, the compiler is wasting 101 vector lanes when choosing a vector length of 128;
- Ideally, one can choose to reduce the vector length to 27, since it is known that this is the exact number of loop operations. However, NVIDIA GPUs have a hardware vector length, also known as **warp size**, of 32;
- Hence, it is possible to choose a hardware configuration wasting only 5 vector lanes instead of 101 using the vector length with the same size of the **warp size**;
- One can use the **device_type(nvidia** clause to perform such optimization only when using a NVIDIA architecture. This modification will not affect the decision of the compiler when using other platforms.

matrix.h

```
29  void allocate_3d_poisson_matrix(matrix &A, int N) {
30    int num_rows=(N+1)*(N+1)*(N+1);
31    int nnz=27*num_rows;
32    A.num_rows=num_rows;
33    A.row_offsets=(unsigned int*)malloc((num_rows+1)*sizeof(unsigned int));
34    A.cols=(unsigned int*)malloc(nnz*sizeof(unsigned int));
35    A.coefs=(double*)malloc(nnz*sizeof(double));

37    int offsets[27];
38    double coefs[27];
39    int zstride=N*N;
40    int ystride=N;

42    int i=0;
43    for(int z=-1;z<=1;z++) {
44      for(int y=-1;y<=1;y++) {
45        for(int x=-1;x<=1;x++) {
46          offsets[i]=zstride*z+ystride*y+x;
47          if(x==0 && y==0 && z==0)
48            coefs[i]=27;
49          else
50            coefs[i]=-1;
51          i++;
52        }
53      }
54    }
```

```
56    nnz=0;
57    for(int i=0;i<num_rows;i++) {
58      A.row_offsets[i]=nnz;
59      for(int j=0;j<27;j++) {
60        int n=i+offsets[j];
61        if(n>=0 && n<num_rows) {
62          A.cols[nnz]=n;
63          A.coefs[nnz]=coefs[j];
64          nnz++;
65        }
66      }
67    }
68
69    A.row_offsets[num_rows]=nnz;
70    A.nnz=nnz;
71  #pragma acc enter data copyin(A)
72  #pragma acc enter data copyin(A.row_offsets[0:num_rows+1],A.cols[0:nnz],A.coefs[0:nnz])
73  }
```

Optimize Loops

Reduce Vector Length

- The inner loop always operates on 27 non-zero elements. Therefore, the compiler is wasting 101 vector lanes when choosing a vector length of 128;
- Ideally, one can choose to reduce the vector length to 27, since it is known that this is the exact number of loop operations. However, NVIDIA GPUs have a hardware vector length, also known as **warp size**, of 32;
- Hence, it is possible to choose a hardware configuration wasting only 5 vector lanes instead of 101 using the vector length with the same size of the **warp size**;
- One can use the **device_type(nvidia** clause to perform such optimization only when using a NVIDIA architecture. This modification will not affect the decision of the compiler when using other platforms.

matrix_functions.h

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30    for(int i=0;i<num_rows;i++) {
31      double sum=0;
32      int row_start=row_offsets[i];
33      int row_end=row_offsets[i+1];
34  #pragma acc loop device_type(nvidia) vector(32)
35      for(int j=row_start;j<row_end;j++) {
36        unsigned int Acol=cols[j];
37        double Acoef=Acoefs[j];
38        double xcoef=xcoefs[Acol];
39        sum+=Acoef*xcoef;
40      }
41      ycoefs[i]=sum;
42    }
43  }
```

Optimize Loops

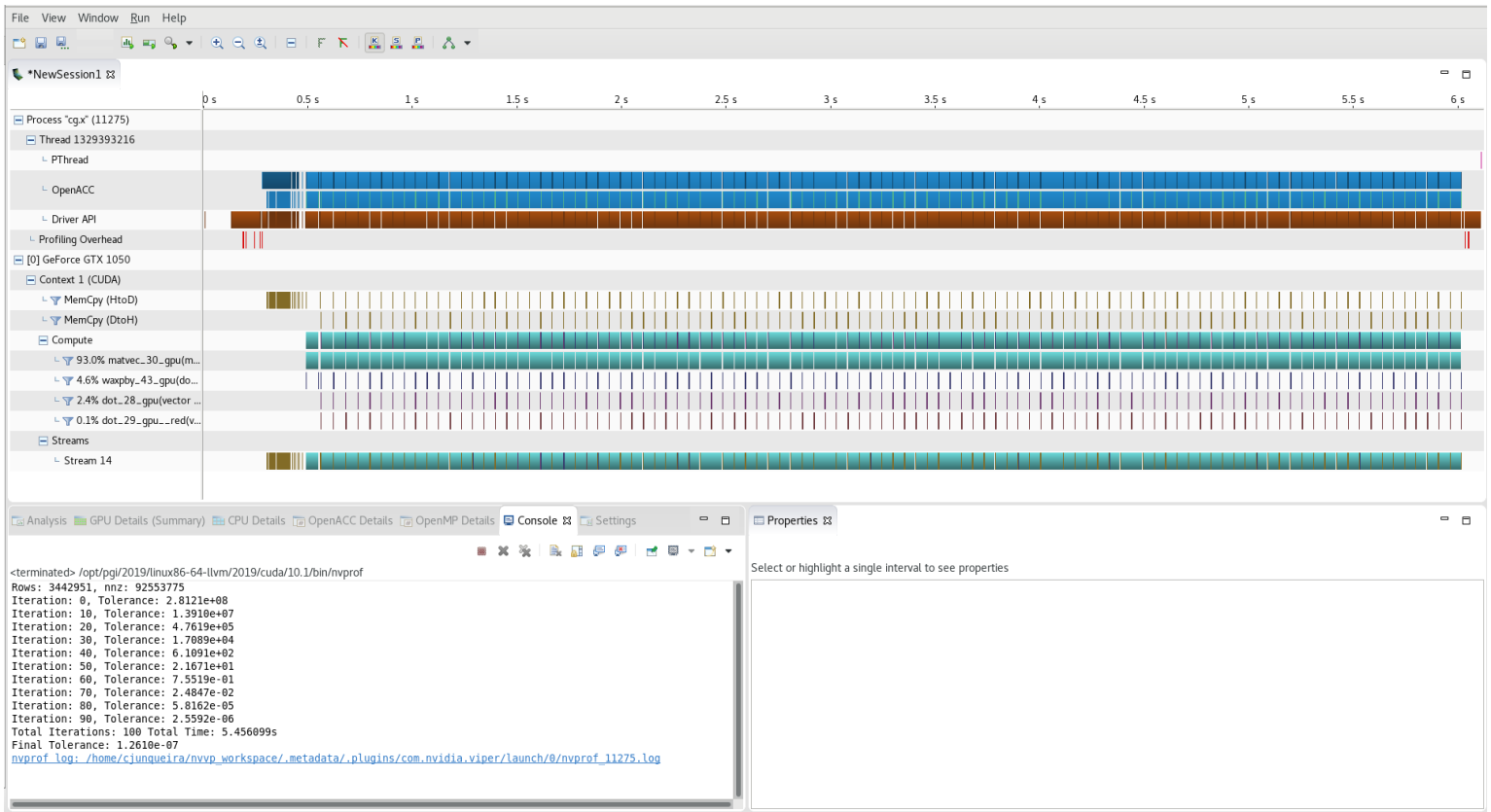
Compiler Feedback

- Compiler uses a vector length of 32 as requested.

```
34      23, include "matrix_functions.h"
35           27, Generating copyin(Acoefs[:A->nnz]) [if not already present]
36               Generating implicit copyin(row_offsets[:num_rows+1]) [if not already present]
37               Generating copyin(cols[:A->nnz]) [if not already present]
38               Generating implicit copyout(ycoefs[:num_rows]) [if not already present]
39               Generating copyin(xcoefs[:x->n]) [if not already present]
40           30, Loop is parallelizable
41               Generating Tesla code
42               30, #pragma acc loop gang /* blockIdx.x */
43               35, #pragma acc loop vector(32) /* threadIdx.x */
44               39, Generating implicit reduction(+:sum)
45           35, Loop is parallelizable
```

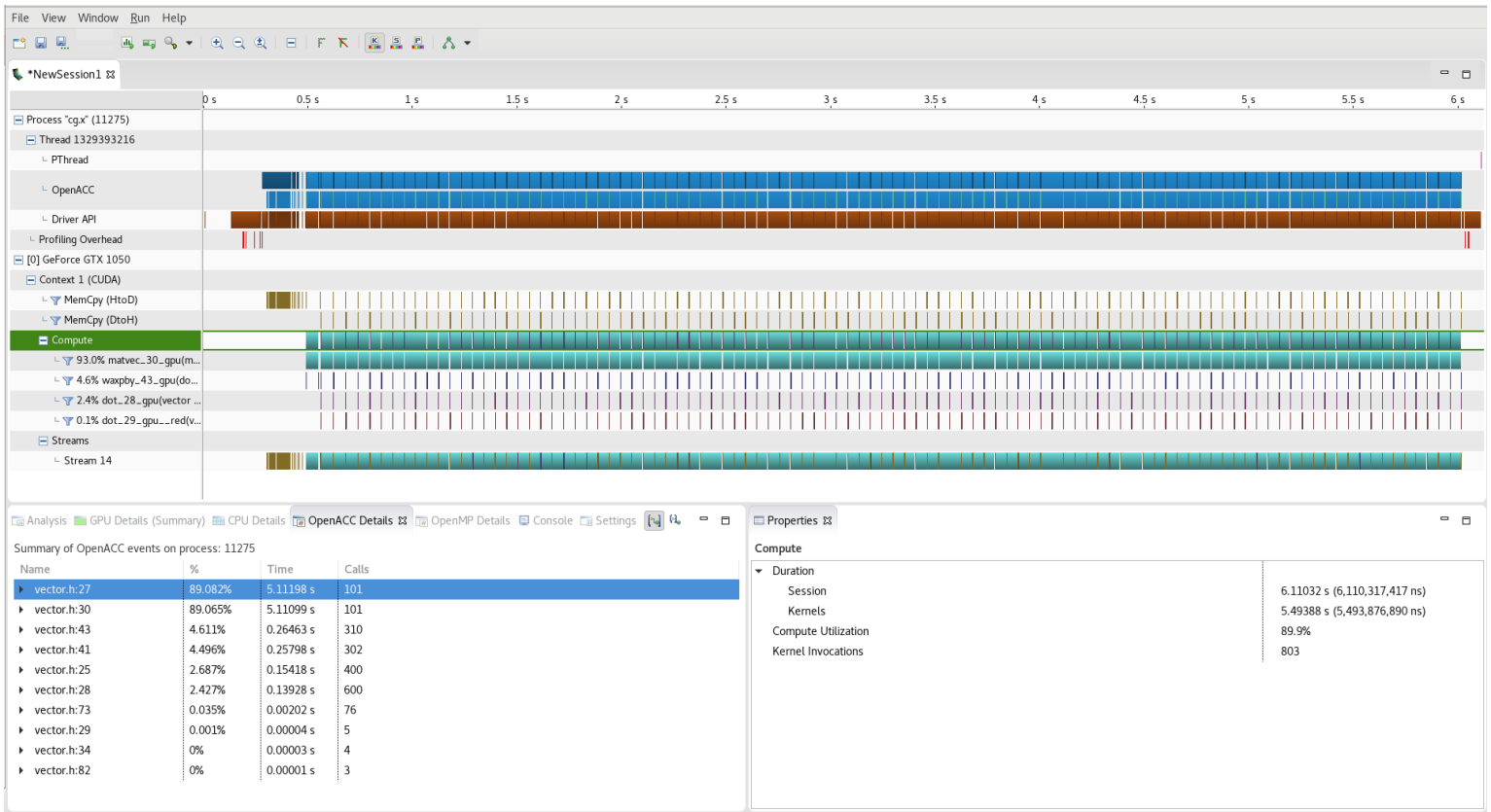
Reduce Vector Length

Profile Timeline



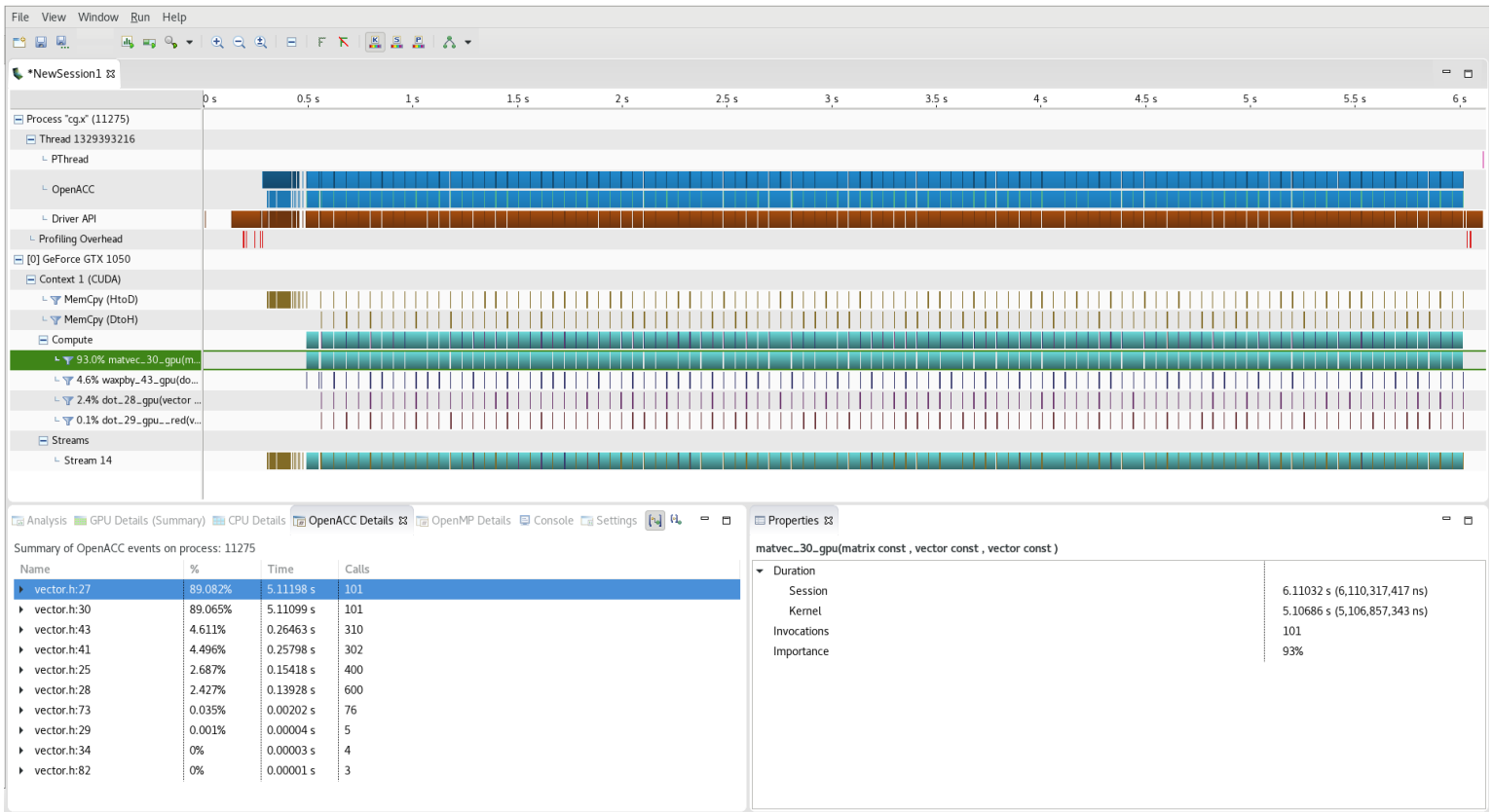
Reduce Vector Length

Profile Timeline



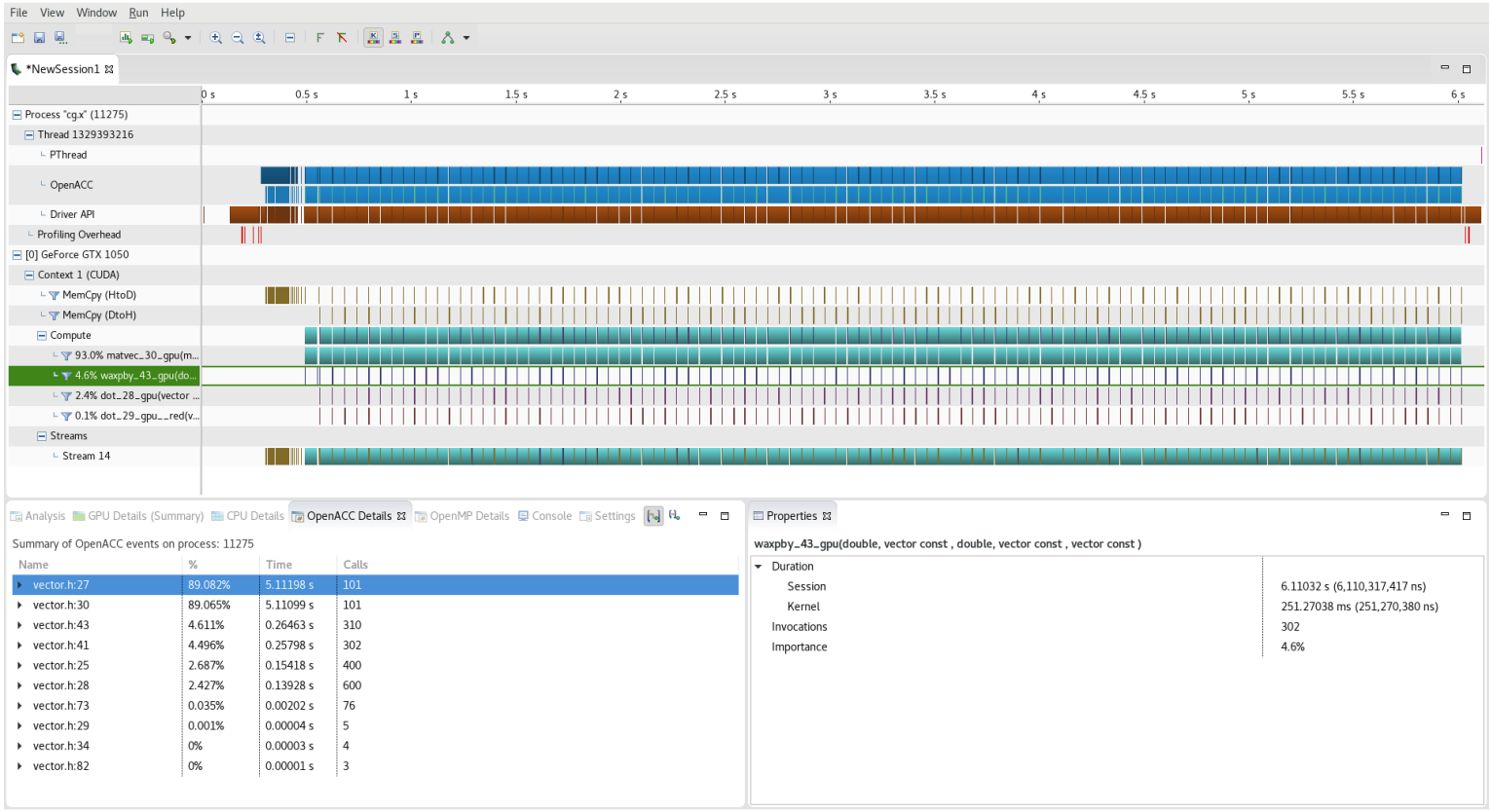
Reduce Vector Length

Profile Timeline



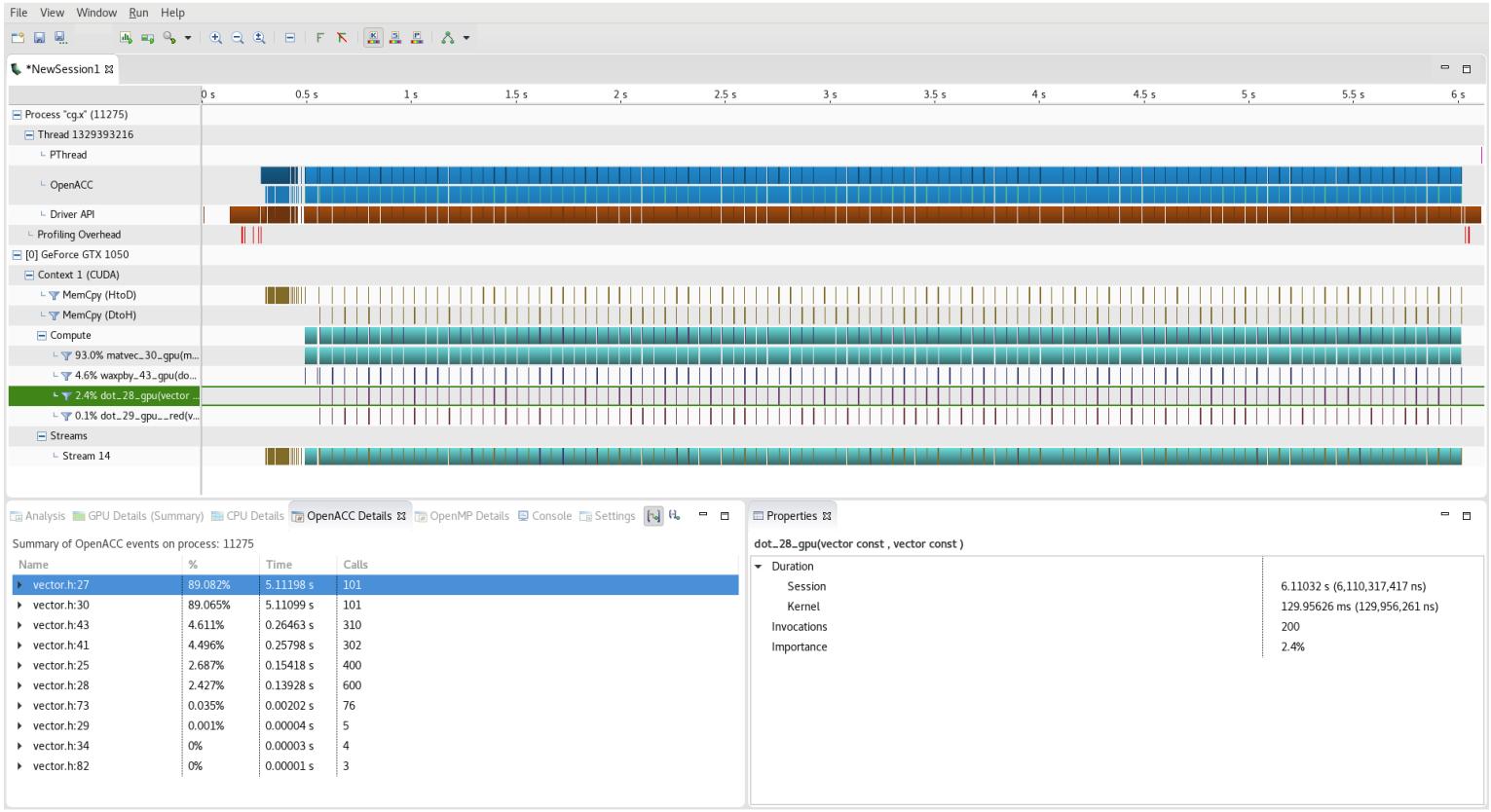
Reduce Vector Length

Profile Timeline



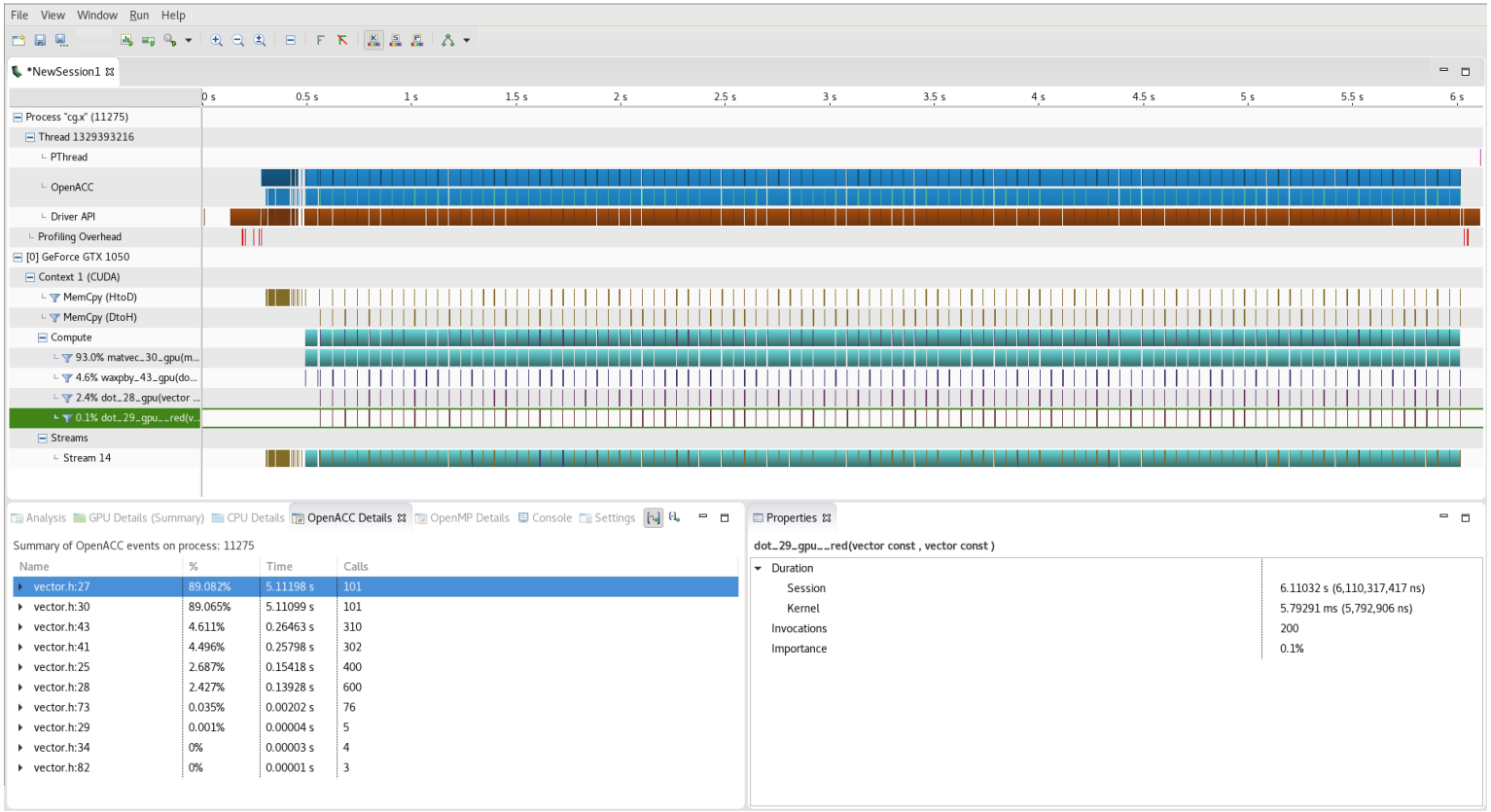
Reduce Vector Length

Profile Timeline



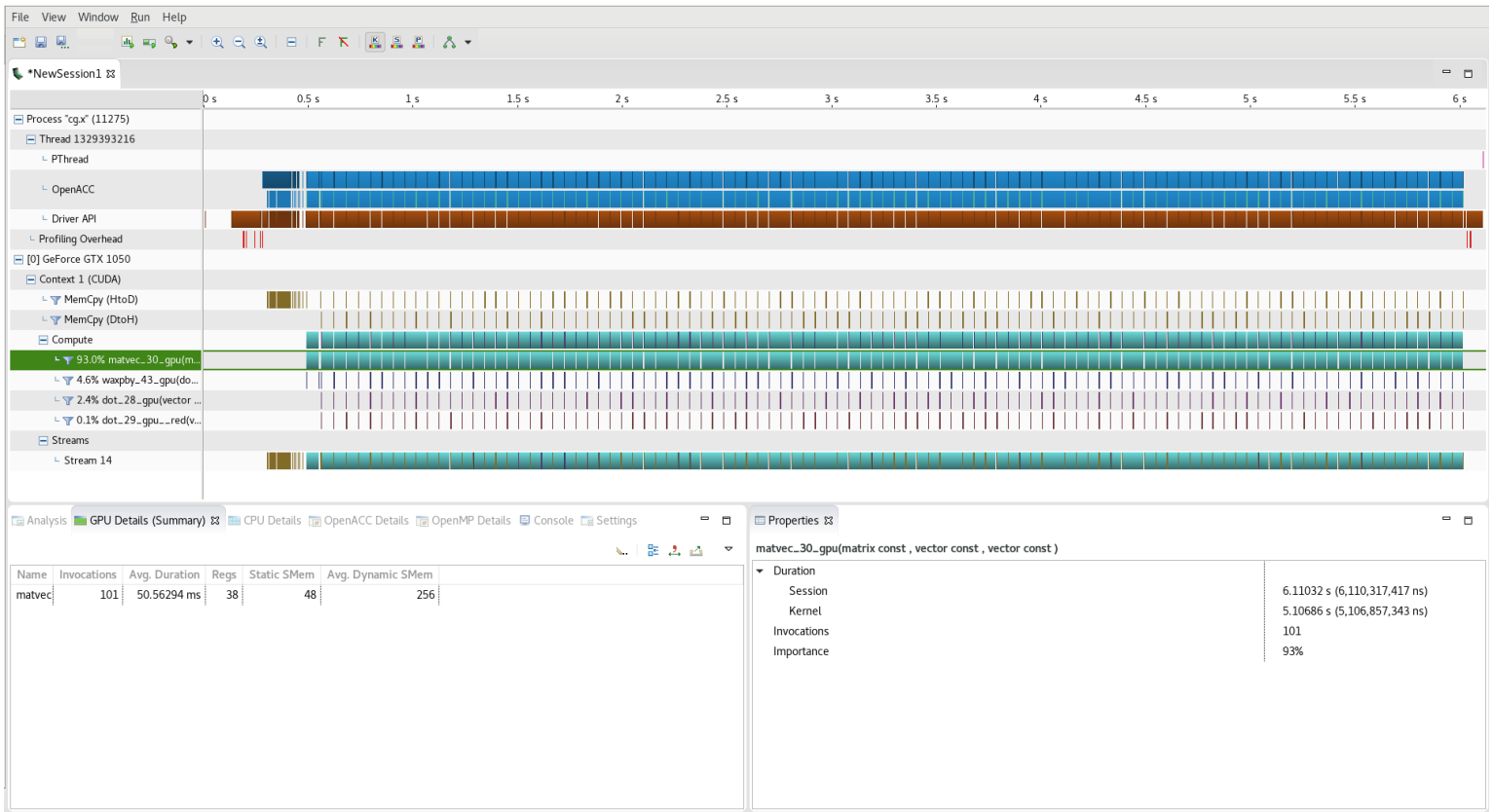
Reduce Vector Length

Profile Timeline



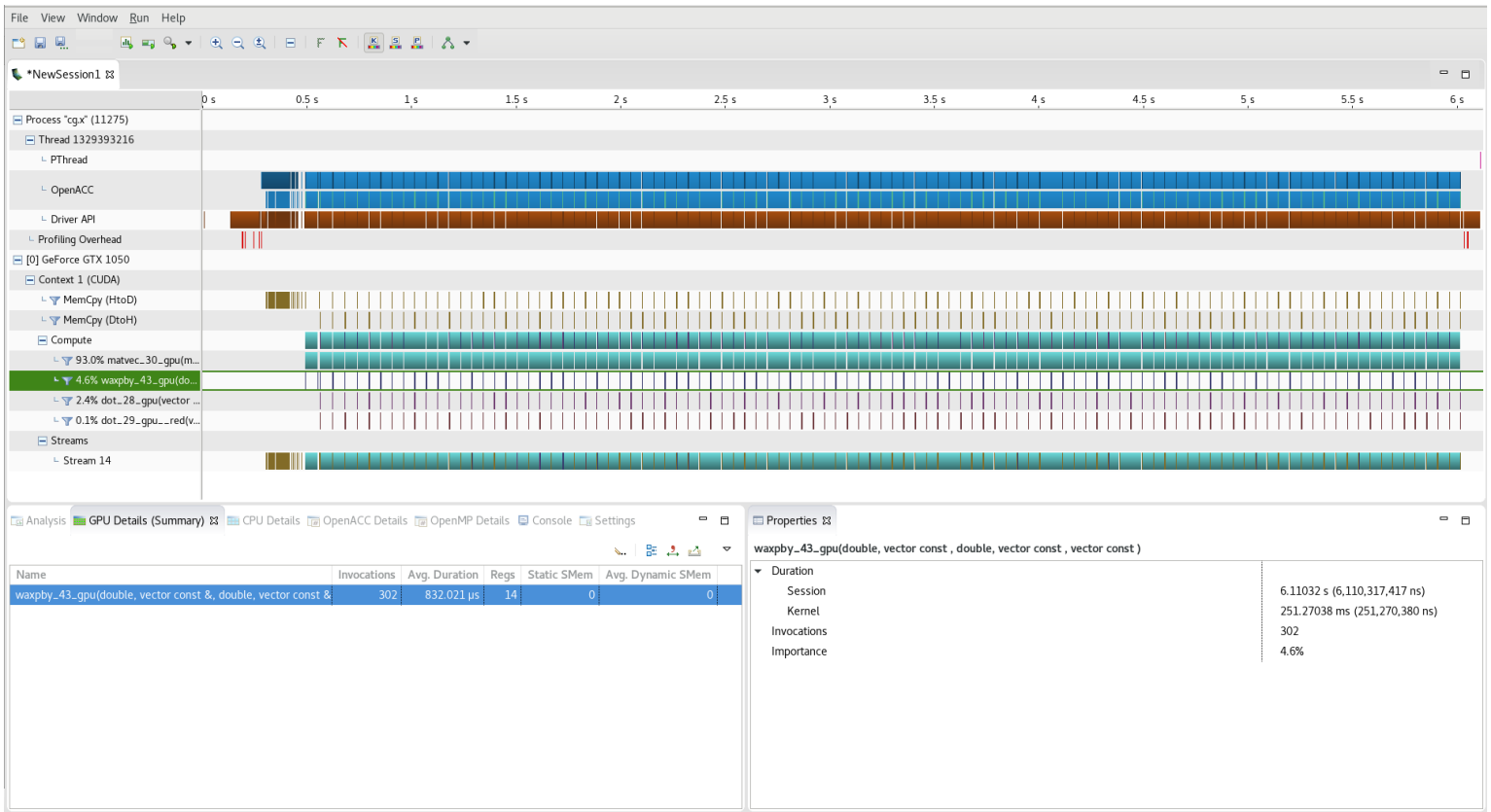
Reduce Vector Length

Profile Timeline



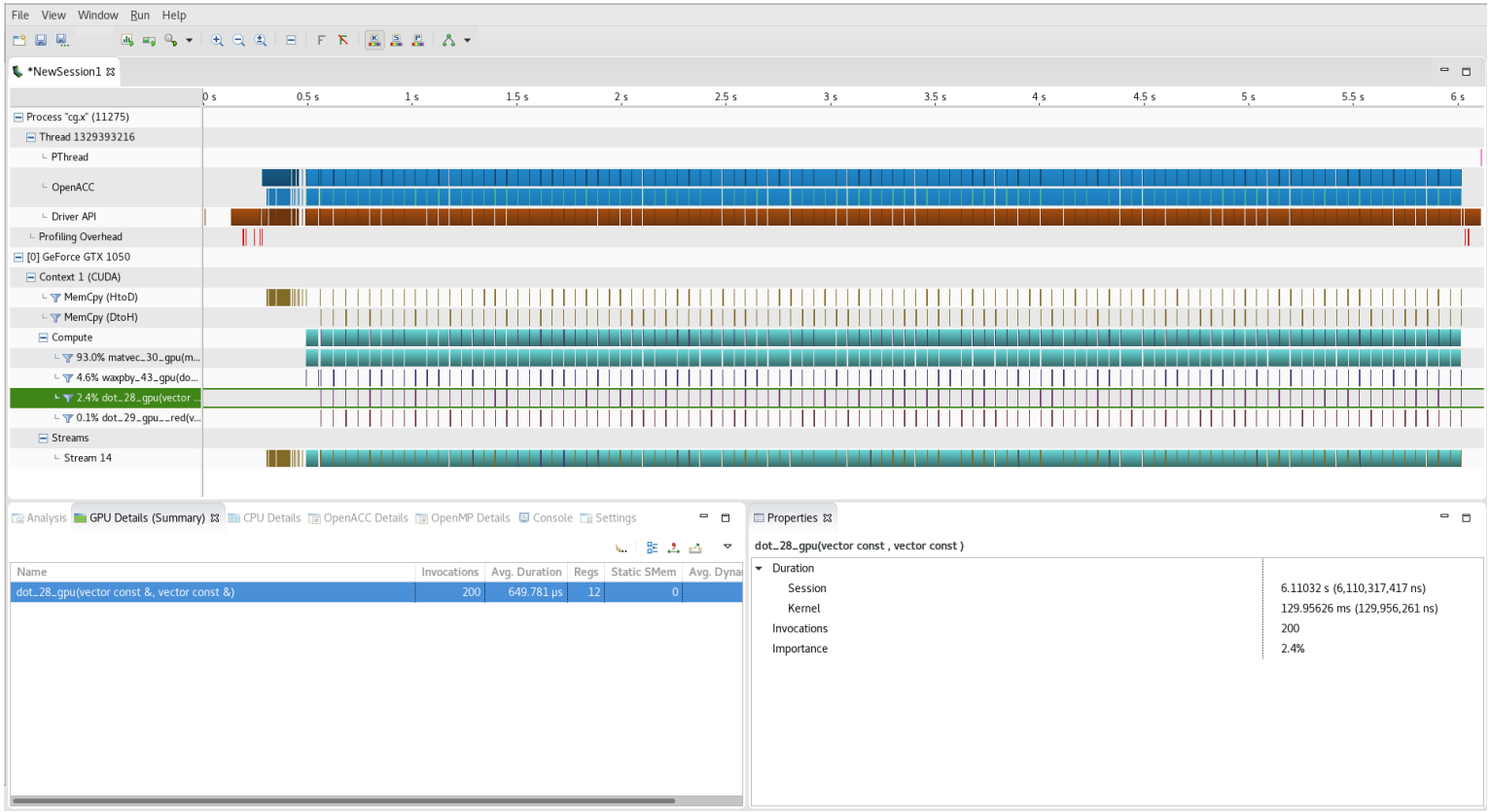
Reduce Vector Length

Profile Timeline



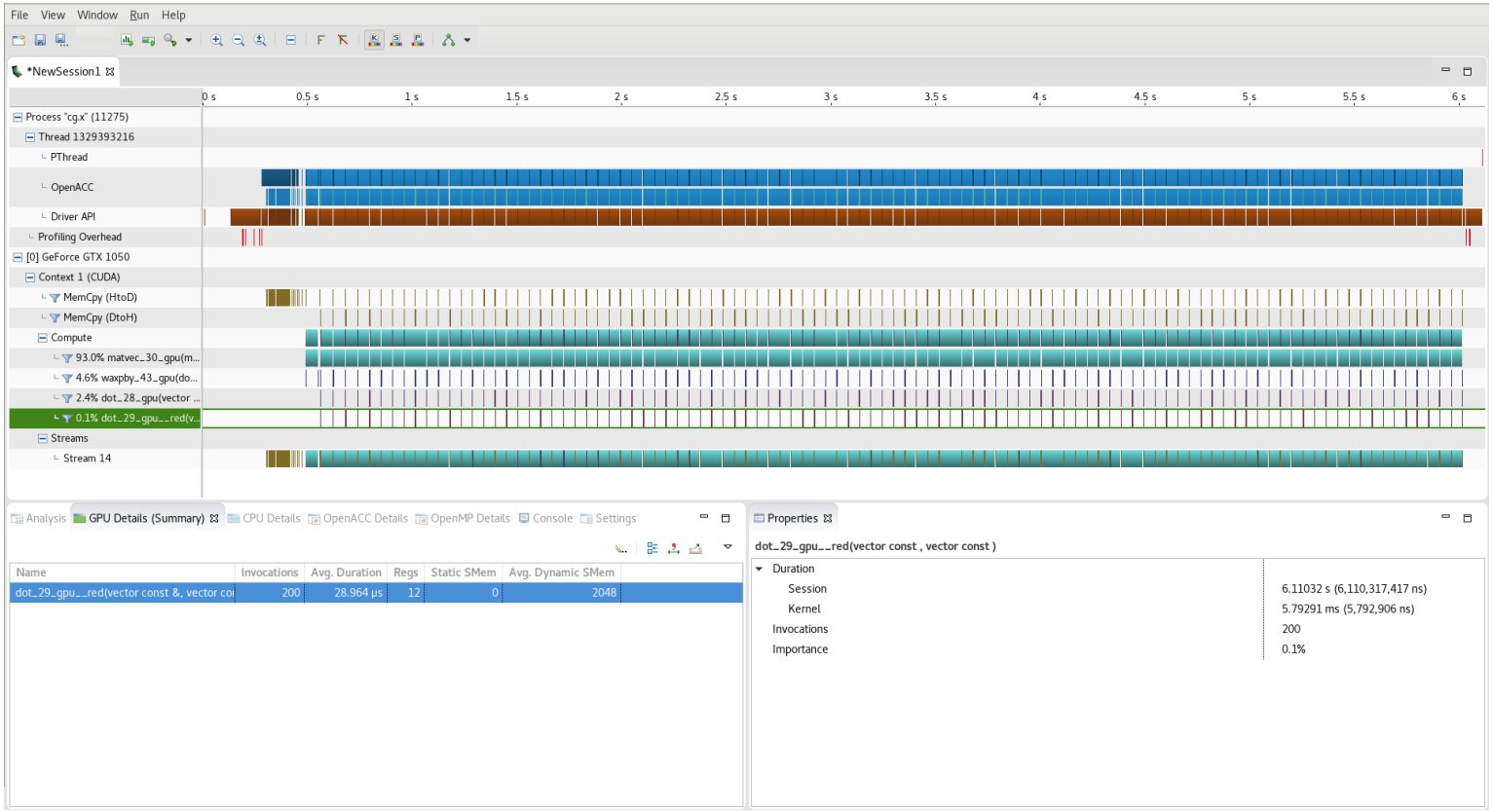
Reduce Vector Length

Profile Timeline



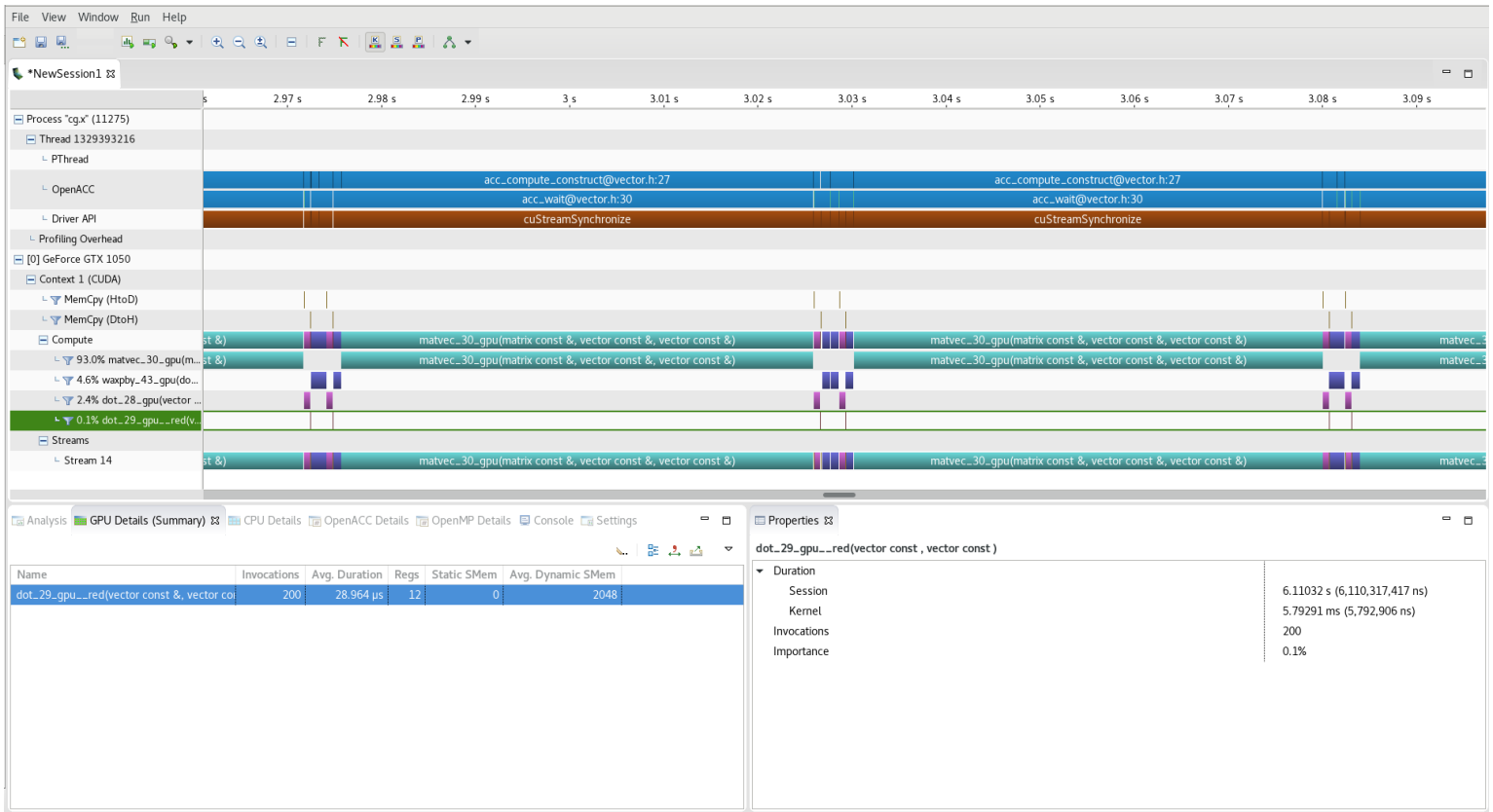
Reduce Vector Length

Profile Timeline



Reduce Vector Length

Profile Timeline



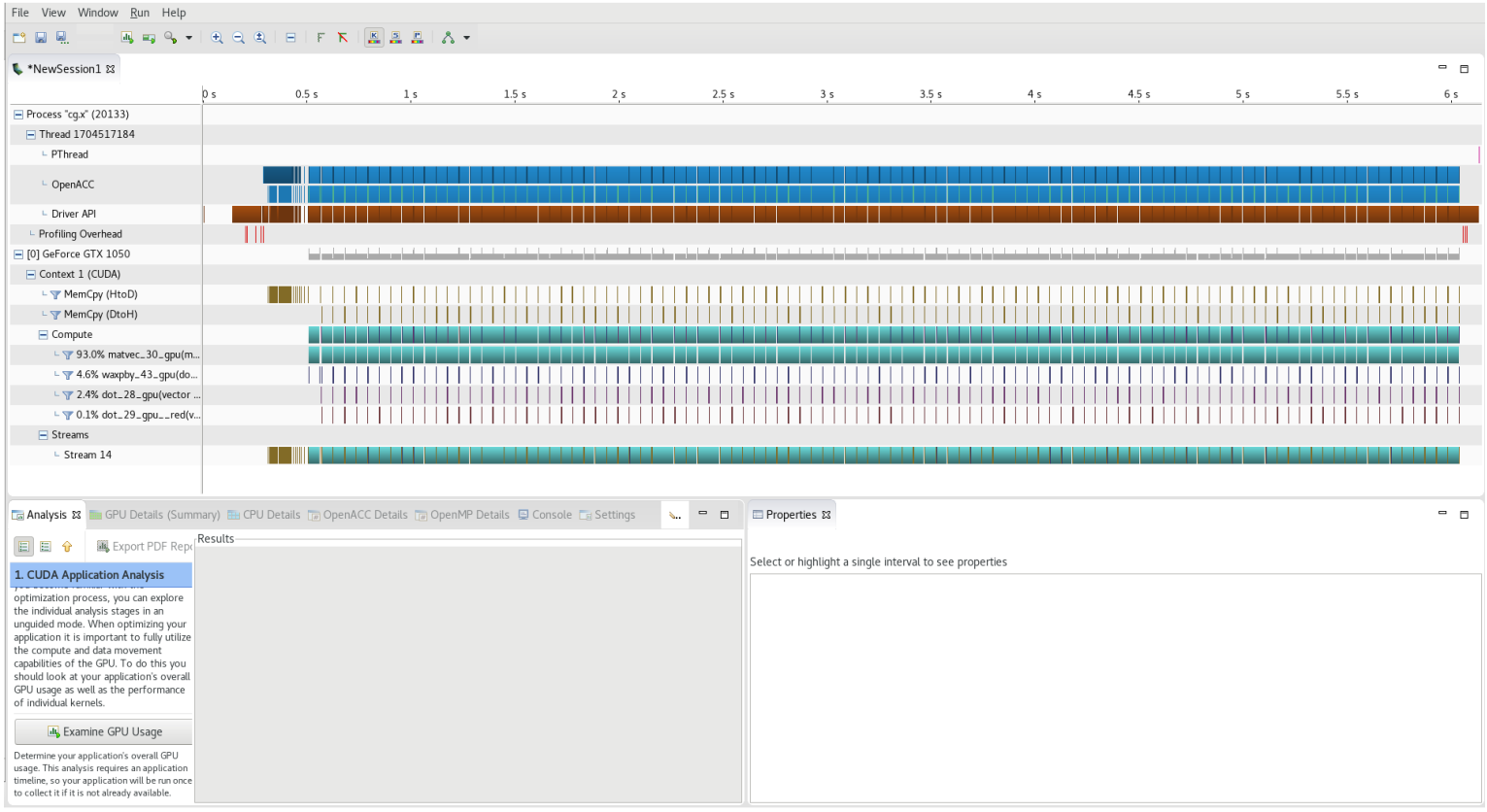
Increase Parallelism

PGProf Guided Analysis Mode

- PGProf guided analysis mode can help to determine the performance limitations;
- After starting a new section, with the version using a vector length of 32 in the **matvec** function, one can select the Analysis tab and follow the suggestions provided by PGProf;

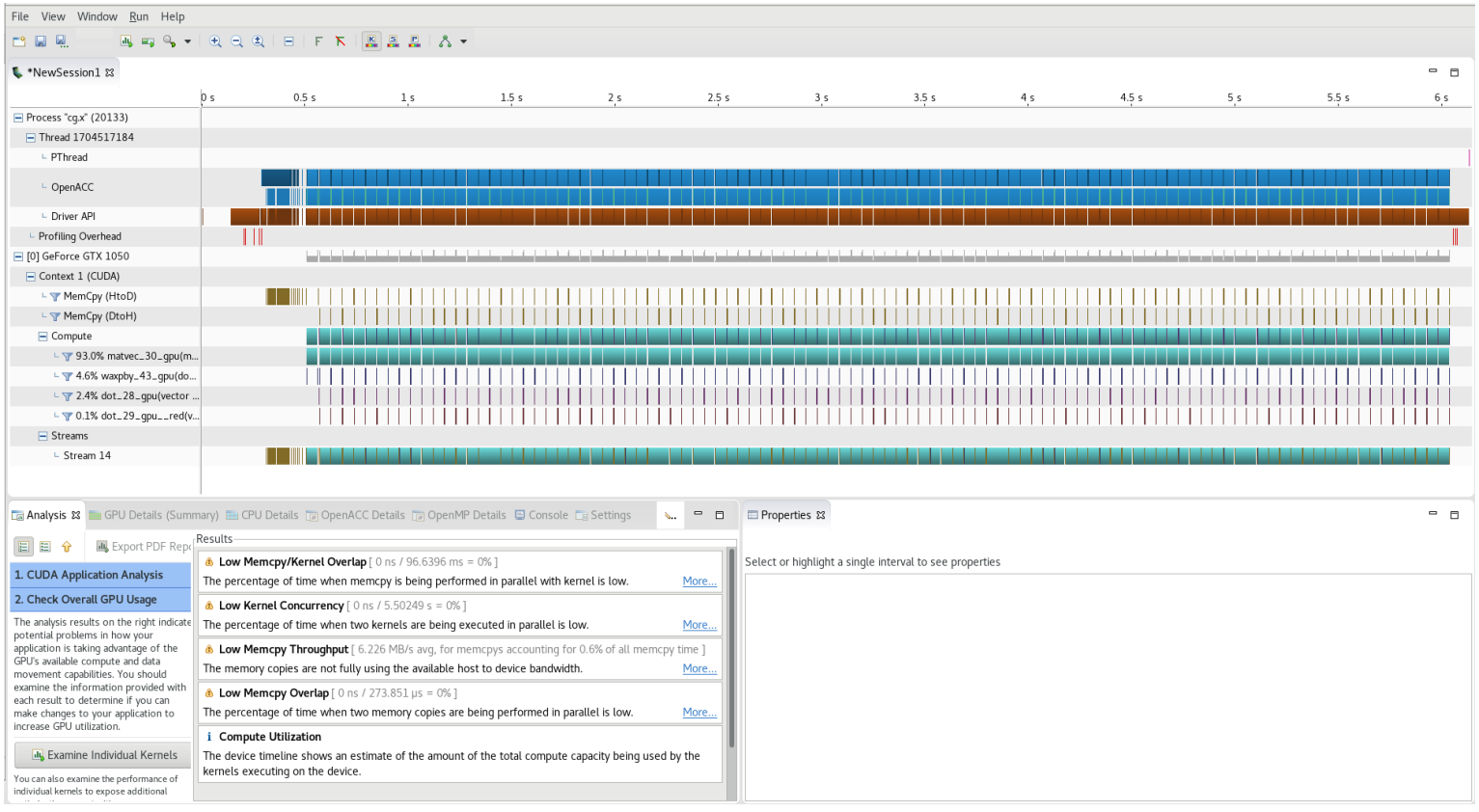
Increase Parallelism

PGProf Guided Analysis Mode



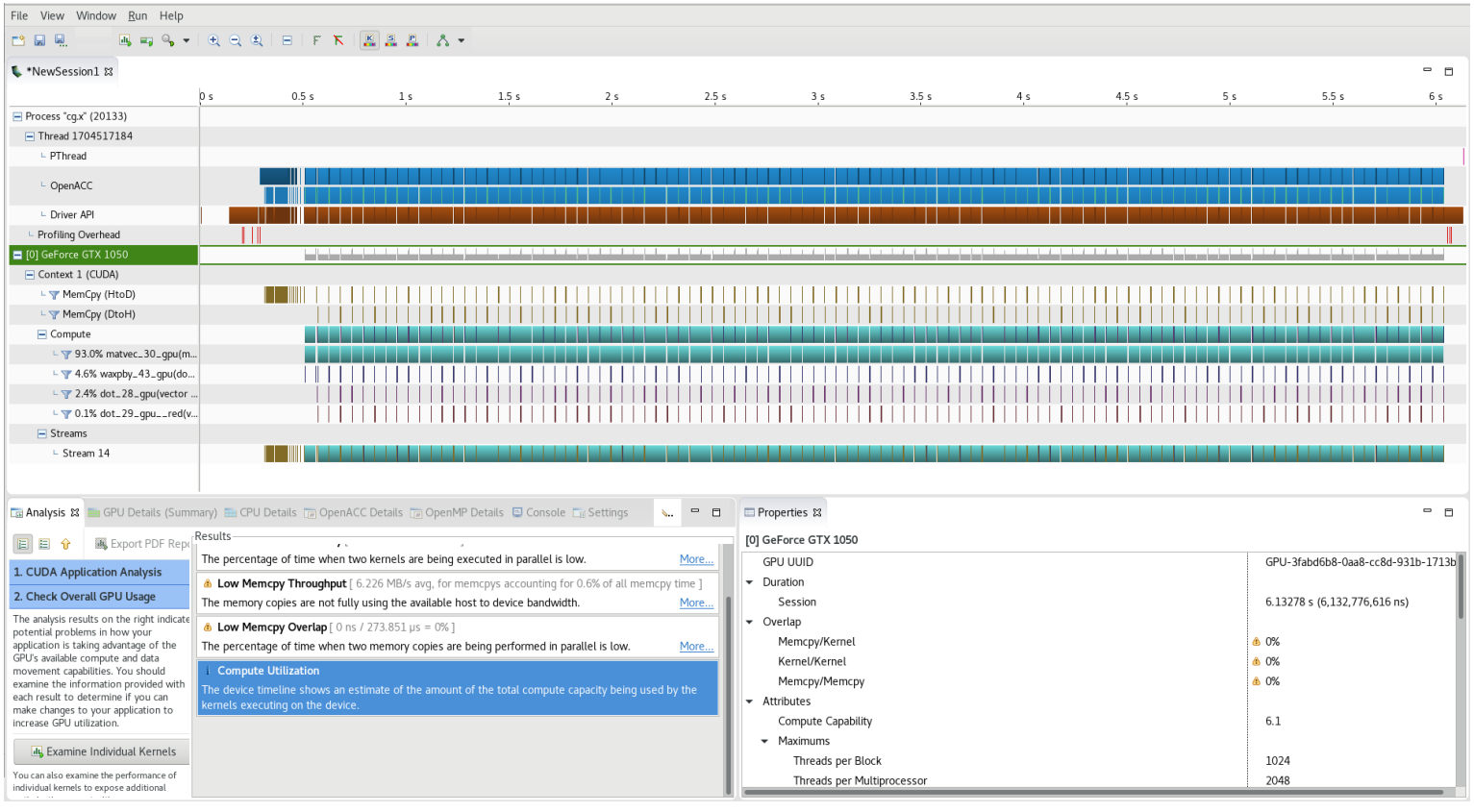
Increase Parallelism

PGProf Guided Analysis Mode



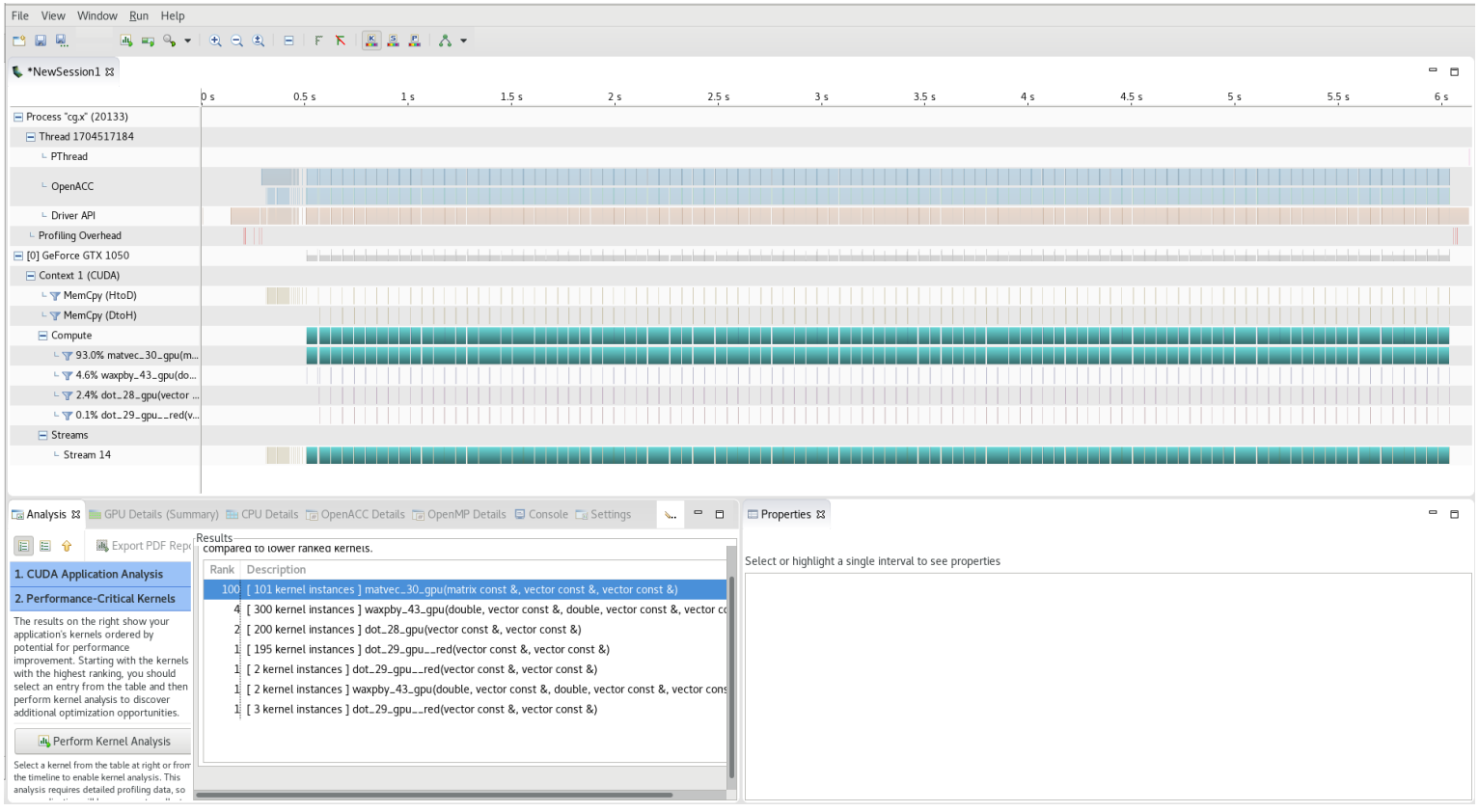
Increase Parallelism

PGProf Guided Analysis Mode



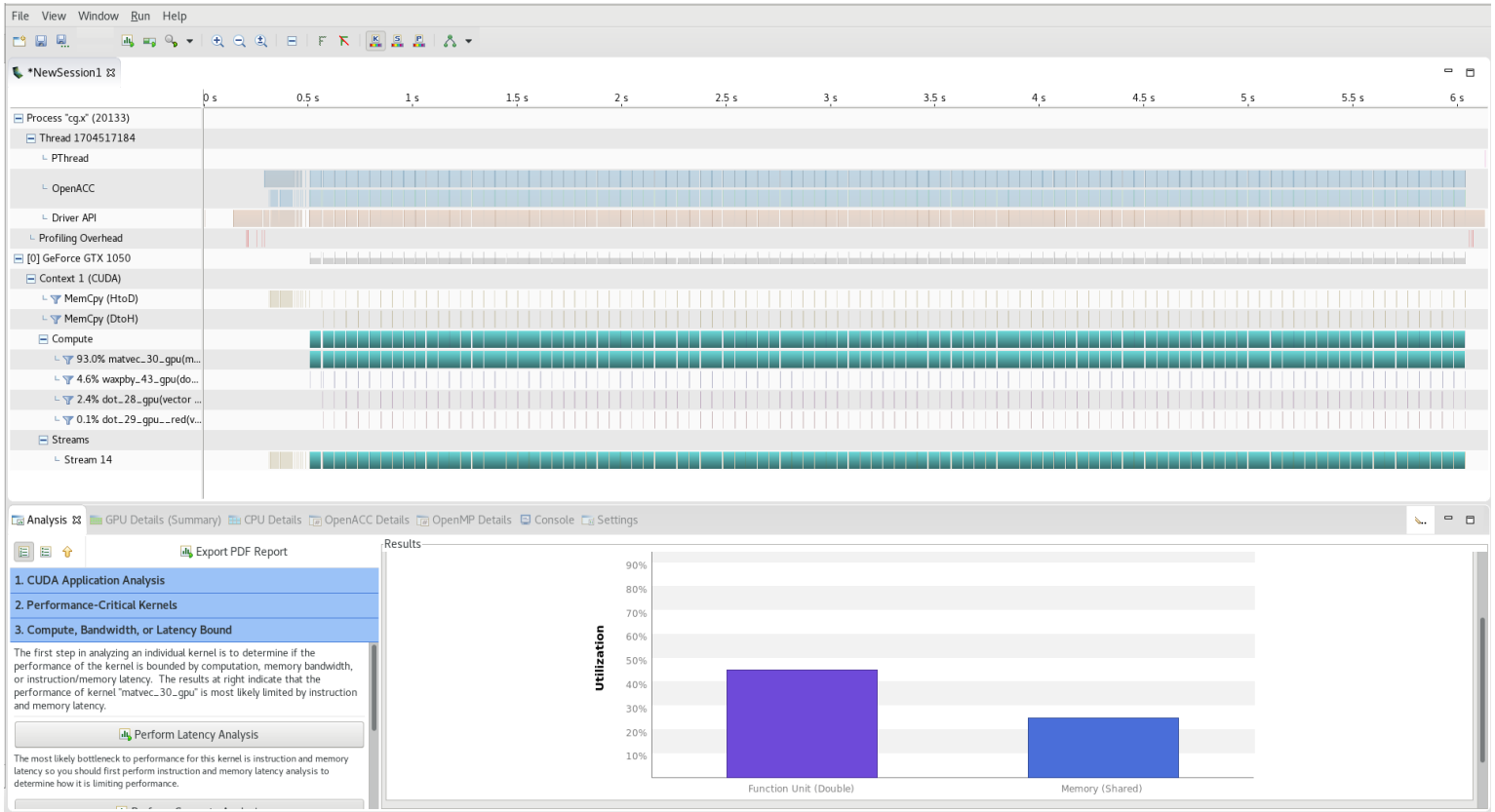
Increase Parallelism

PGProf Guided Analysis Mode



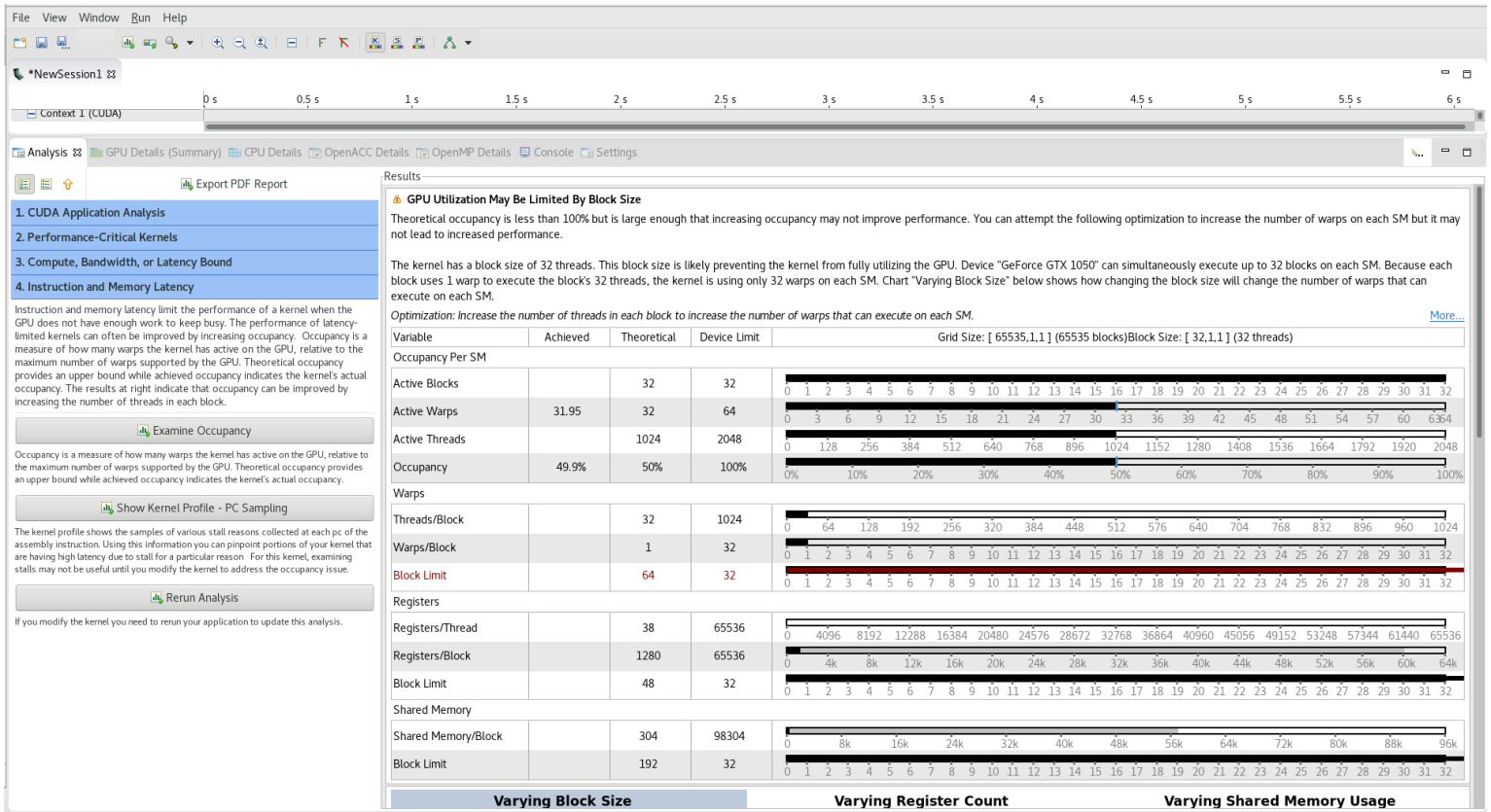
Increase Parallelism

PGProf Guided Analysis Mode



Increase Parallelism

PGProf Guided Analysis Mode



Increase Parallelism

PGProf Guided Analysis Mode

- One can notice the *occupancy* as a limiting factor;
- *Occupancy* is a term used by NVIDIA to describe how well the GPU is saturated compared to how much work it could theoretically run.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [32,1,1] (32 threads)
Occupancy Per SM				
Active Blocks		32	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Active Warps	31.95	32	64	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 6364
Active Threads		1024	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	49.9%	50%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		32	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		1	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block Limit		64	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Registers				
Registers/Thread		38	65536	0 4096 8192 12288 16384 20480 24576 28672 32768 36864 40960 45056 49152 53248 57344 61440 65536
Registers/Block		1280	65536	0 4k 8k 12k 16k 20k 24k 28k 32k 36k 40k 44k 48k 52k 56k 60k 64k
Block Limit		48	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Shared Memory				
Shared Memory/Block		304	98304	0 8k 16k 24k 32k 40k 48k 56k 64k 72k 80k 88k 96k
Block Limit		192	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Increase Parallelism

PGProf Guided Analysis Mode

- NVIDIA GPUs are comprised of 1 or more *streaming multiprocessors*, commonly referred as *SM*s;
- An SM can manage up to 2048 concurrent threads, although not all of these threads will be actively running at the same time.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [32,1,1] (32 threads)
Occupancy Per SM				
Active Blocks		32	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Active Warps	31.95	32	64	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 6364
Active Threads		1024	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	49.9%	50%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		32	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		1	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block Limit		64	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Registers				
Registers/Thread		38	65536	0 4096 8192 12288 16384 20480 24576 28672 32768 36864 40960 45056 49152 53248 57344 61440 65536
Registers/Block		1280	65536	0 4k 8k 12k 16k 20k 24k 28k 32k 36k 40k 44k 48k 52k 56k 60k 64k
Block Limit		48	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Shared Memory				
Shared Memory/Block		304	98304	0 8k 16k 24k 32k 40k 48k 56k 64k 72k 80k 88k 96k
Block Limit		192	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Increase Parallelism

PGProf Guided Analysis Mode

- The profiler output is showing that of the possible 2048 active threads, the SM only has 1024, resulting in 50% occupancy.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [32,1,1] (32 threads)
Occupancy Per SM				
Active Blocks		32	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Active Warps	31.95	32	64	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 6364
Active Threads		1024	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	49.9%	50%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		32	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		1	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block Limit		64	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Registers				
Registers/Thread		38	65536	0 4096 8192 12288 16384 20480 24576 28672 32768 36864 40960 45056 49152 53248 57344 61440 65536
Registers/Block		1280	65536	0 4k 8k 12k 16k 20k 24k 28k 32k 36k 40k 44k 48k 52k 56k 60k 64k
Block Limit		48	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Shared Memory				
Shared Memory/Block		304	98304	0 8k 16k 24k 32k 40k 48k 56k 64k 72k 80k 88k 96k
Block Limit		192	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Increase Parallelism

PGProf Guided Analysis Mode

- The red line shows that the SM can manage at most 32 threadblocks, which equate to OpenACC gangs, but it would require 64 to achieve full occupancy due the size of the gang, which one can see from the *Thread/Block* line has only 32 threads. It is possible to say that, in this case, there is only one worker per gang.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [65535,1,1] (65535 blocks)Block Size: [32,1,1] (32 threads)
Occupancy Per SM				
Active Blocks		32	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Active Warps	31.95	32	64	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 63 64
Active Threads		1024	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	49.9%	50%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		32	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		1	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block Limit		64	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Registers				
Registers/Thread		38	65536	0 4096 8192 12288 16384 20480 24576 28672 32768 36864 40960 45056 49152 53248 57344 61440 65536
Registers/Block		1280	65536	0 4k 8k 12k 16k 20k 24k 28k 32k 36k 40k 44k 48k 52k 56k 60k 64k
Block Limit		48	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Shared Memory				
Shared Memory/Block		304	98304	0 8k 16k 24k 32k 40k 48k 56k 64k 72k 80k 88k 96k
Block Limit		192	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

Increase Parallelism

PGProf Guided Analysis Mode

- It is necessary to introduce more parallelism to better occupy the GPU. Increasing the parallelism can be achieved increasing the vector length or increasing the number of workers;
- The data structure limits the vector length to 32, as previously discussed. Therefore, the other solution is to better occupy the GPUs using more workers;
- The total work within the GPU threadblock can be thought as the number of *worker* $\times$ *vector length*, so, for this case, the most number of worker is 32, because $32 \times 32 = 1024$ GPU threads.

Architecture Restrictions - Using GP107/GTX1050 (Pascal Family)

- Number of gangs is restricted to $2^{31} - 1$;
- **The 2-D thread-grid size ($nb_{workers} * vector_{length}$) is limited to 1024;**
- Due to register limitations the size of the grid should be less than or equal to 256 if the programmer wants to be sure that a kernel can be launched;
- The size of a worker ($vector_{length}$) should be multiple of 32;
- **PGI limitation:** In a kernel that contains calls to external subroutines (not seq), the size of a worker is set at to 32.

Increase Parallelism

Increase of Number of Workers

- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30  #pragma acc loop device_type(nvidia) gang worker(2)
31    for(int i=0;i<num_rows;i++) {
32      double sum=0;
33      int row_start=row_offsets[i];
34      int row_end=row_offsets[i+1];
35  #pragma acc loop device_type(nvidia) vector(32)
36      for(int j=row_start;j<row_end;j++) {
37        unsigned int Acol=cols[j];
38        double Acoef=Acoefs[j];
39        double xcoef=xcoefs[Acol];
40        sum+=Acoef*xcoef;
41      }
42      ycoefs[i]=sum;
43    }
44  }
```

Increase Parallelism

Increase of Number of Workers

- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30  #pragma acc loop device_type(nvidia) gang worker(4)
31    for(int i=0;i<num_rows;i++) {
32      double sum=0;
33      int row_start=row_offsets[i];
34      int row_end=row_offsets[i+1];
35  #pragma acc loop device_type(nvidia) vector(32)
36      for(int j=row_start;j<row_end;j++) {
37        unsigned int Acol=cols[j];
38        double Acoef=Acoefs[j];
39        double xcoef=xcoefs[Acol];
40        sum+=Acoef*xcoef;
41      }
42      ycoefs[i]=sum;
43    }
44  }
```

Increase Parallelism

Increase of Number of Workers

- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30  #pragma acc loop device_type(nvidia) gang worker(8)
31    for(int i=0;i<num_rows;i++) {
32      double sum=0;
33      int row_start=row_offsets[i];
34      int row_end=row_offsets[i+1];
35  #pragma acc loop device_type(nvidia) vector(32)
36      for(int j=row_start;j<row_end;j++) {
37        unsigned int Acol=cols[j];
38        double Acoef=Acoefs[j];
39        double xcoef=xcoefs[Acol];
40        sum+=Acoef*xcoef;
41      }
42      ycoefs[i]=sum;
43    }
44  }
```

Increase Parallelism

Increase of Number of Workers

- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30  #pragma acc loop device_type(nvidia) gang worker(16)
31    for(int i=0;i<num_rows;i++) {
32      double sum=0;
33      int row_start=row_offsets[i];
34      int row_end=row_offsets[i+1];
35  #pragma acc loop device_type(nvidia) vector(32)
36      for(int j=row_start;j<row_end;j++) {
37        unsigned int Acol=cols[j];
38        double Acoef=Acoefs[j];
39        double xcoef=xcoefs[Acol];
40        sum+=Acoef*xcoef;
41      }
42      ycoefs[i]=sum;
43    }
44  }
```

Increase Parallelism

Increase of Number of Workers

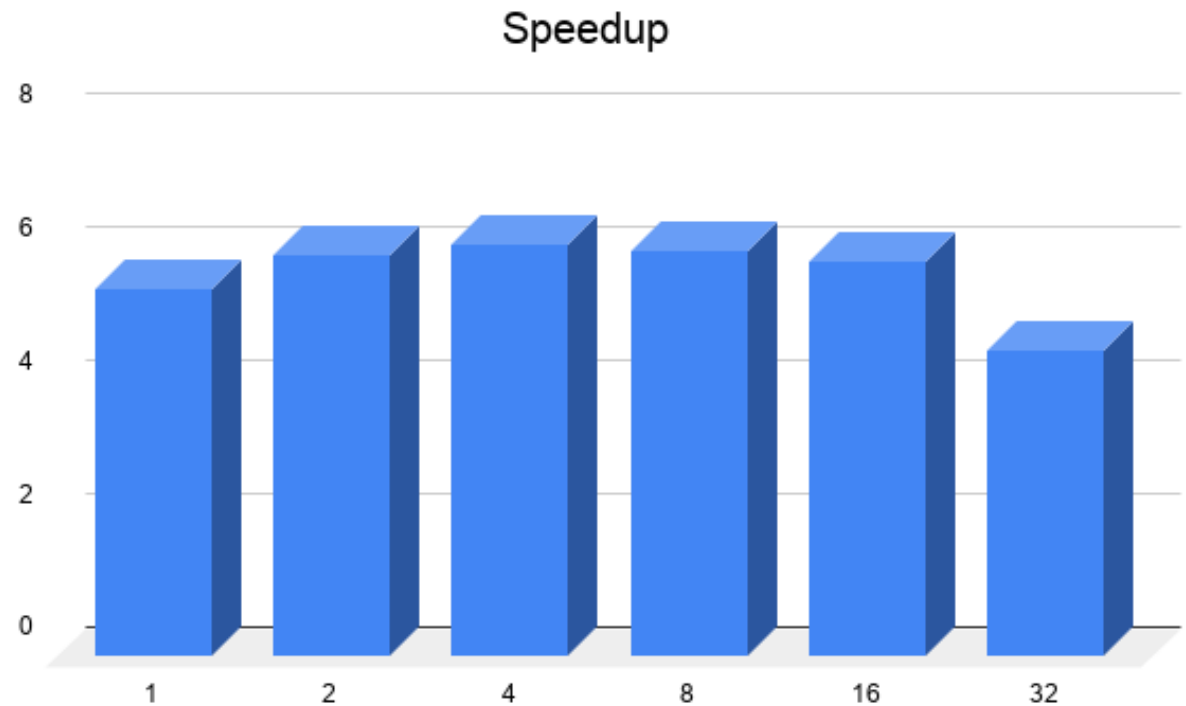
- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.

```
20  void matvec(const matrix& A, const vector& x, const vector &y) {
21
22    unsigned int num_rows=A.num_rows;
23    unsigned int *restrict row_offsets=A.row_offsets;
24    unsigned int *restrict cols=A.cols;
25    double *restrict Acoefs=A.coefs;
26    double *restrict xcoefs=x.coefs;
27    double *restrict ycoefs=y.coefs;
28
29  #pragma acc kernels copyin(cols[0:A.nnz],Acoefs[0:A.nnz],xcoefs[0:x.n])
30  #pragma acc loop device_type(nvidia) gang worker(32)
31    for(int i=0;i<num_rows;i++) {
32      double sum=0;
33      int row_start=row_offsets[i];
34      int row_end=row_offsets[i+1];
35  #pragma acc loop device_type(nvidia) vector(32)
36      for(int j=row_start;j<row_end;j++) {
37        unsigned int Acol=cols[j];
38        double Acoef=Acoefs[j];
39        double xcoef=xcoefs[Acol];
40        sum+=Acoef*xcoef;
41      }
42      ycoefs[i]=sum;
43    }
44  }
```

Increase Parallelism

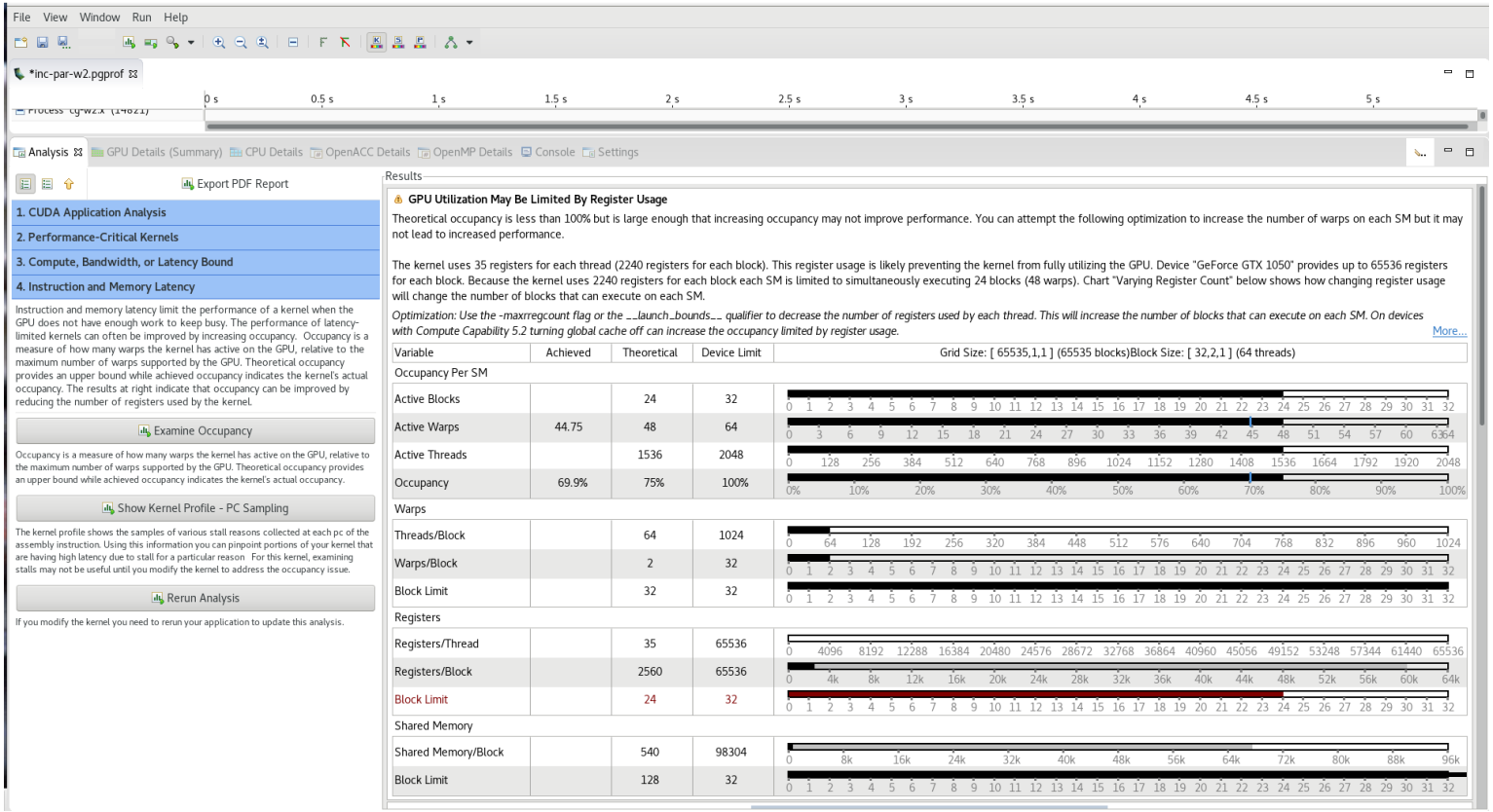
Increase of Number of Workers

- One can increase the number of workers to evaluate the speedups;
- Nb. workers: 2, 4, 8, 16, 32.



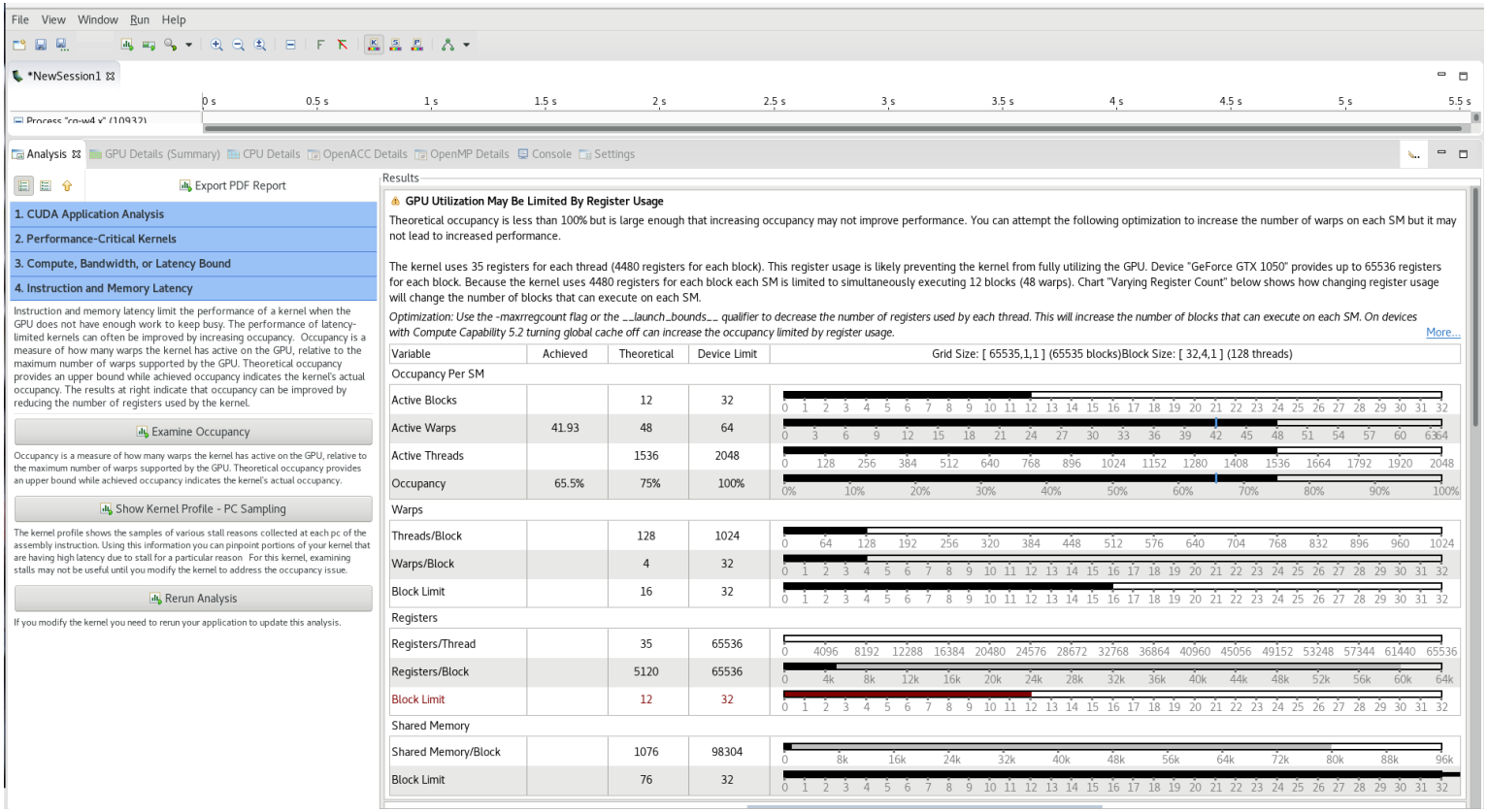
Increase Parallelism

PGProf Guided Analysis Mode - Nb. Workers = 2



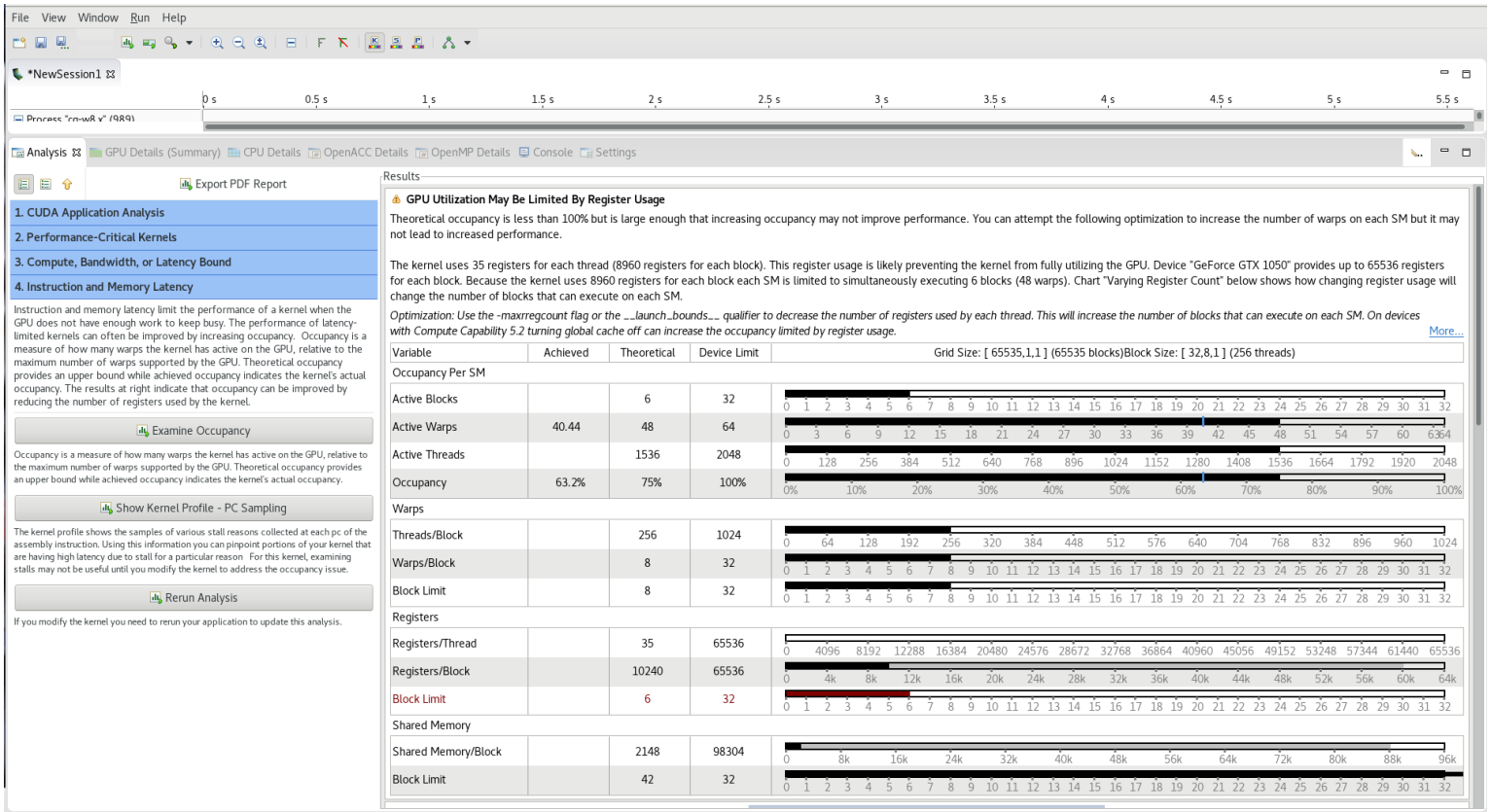
Increase Parallelism

PGProf Guided Analysis Mode - Nb. Workers = 4



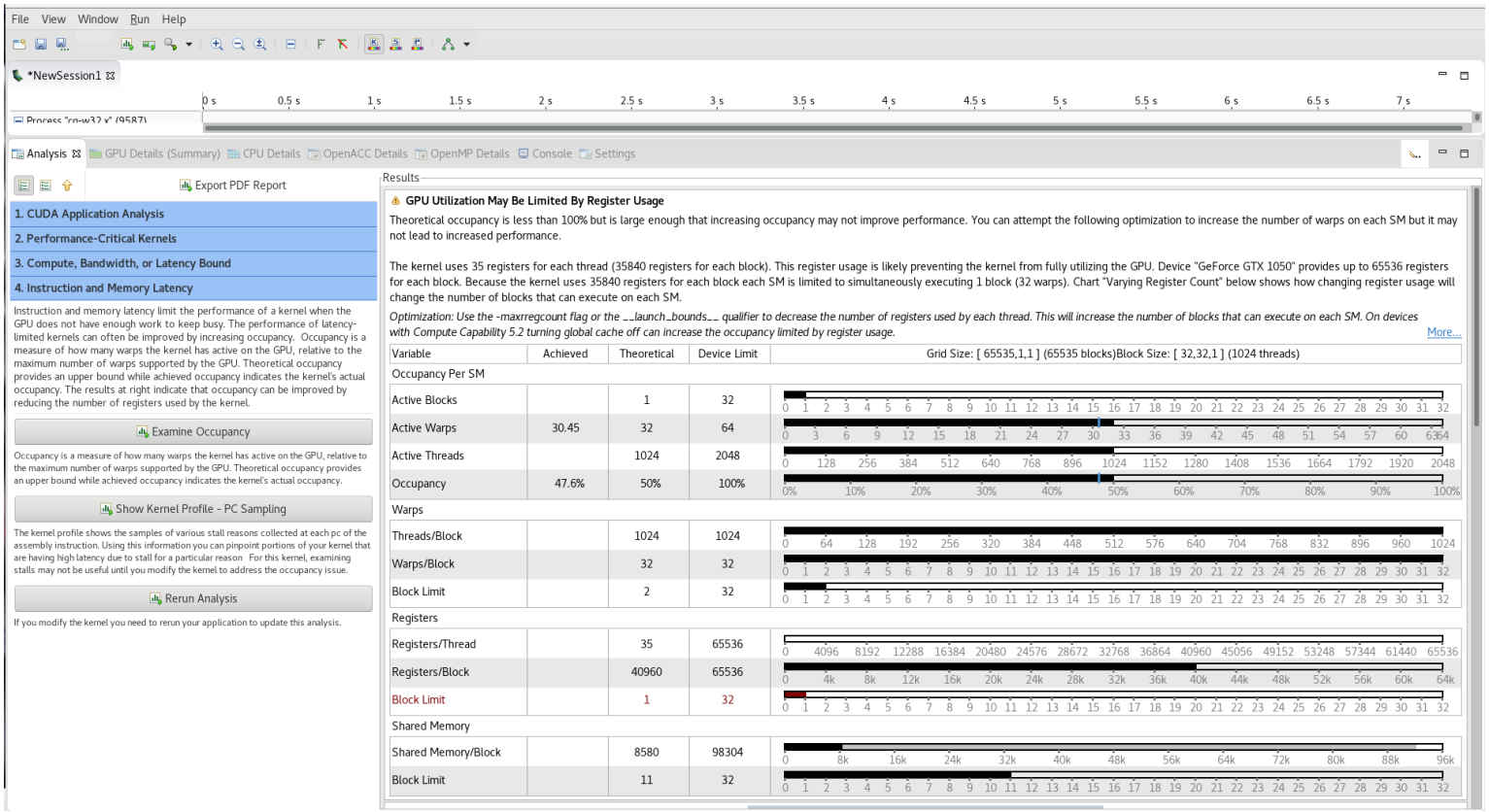
Increase Parallelism

PGProf Guided Analysis Mode - Nb. Workers = 8



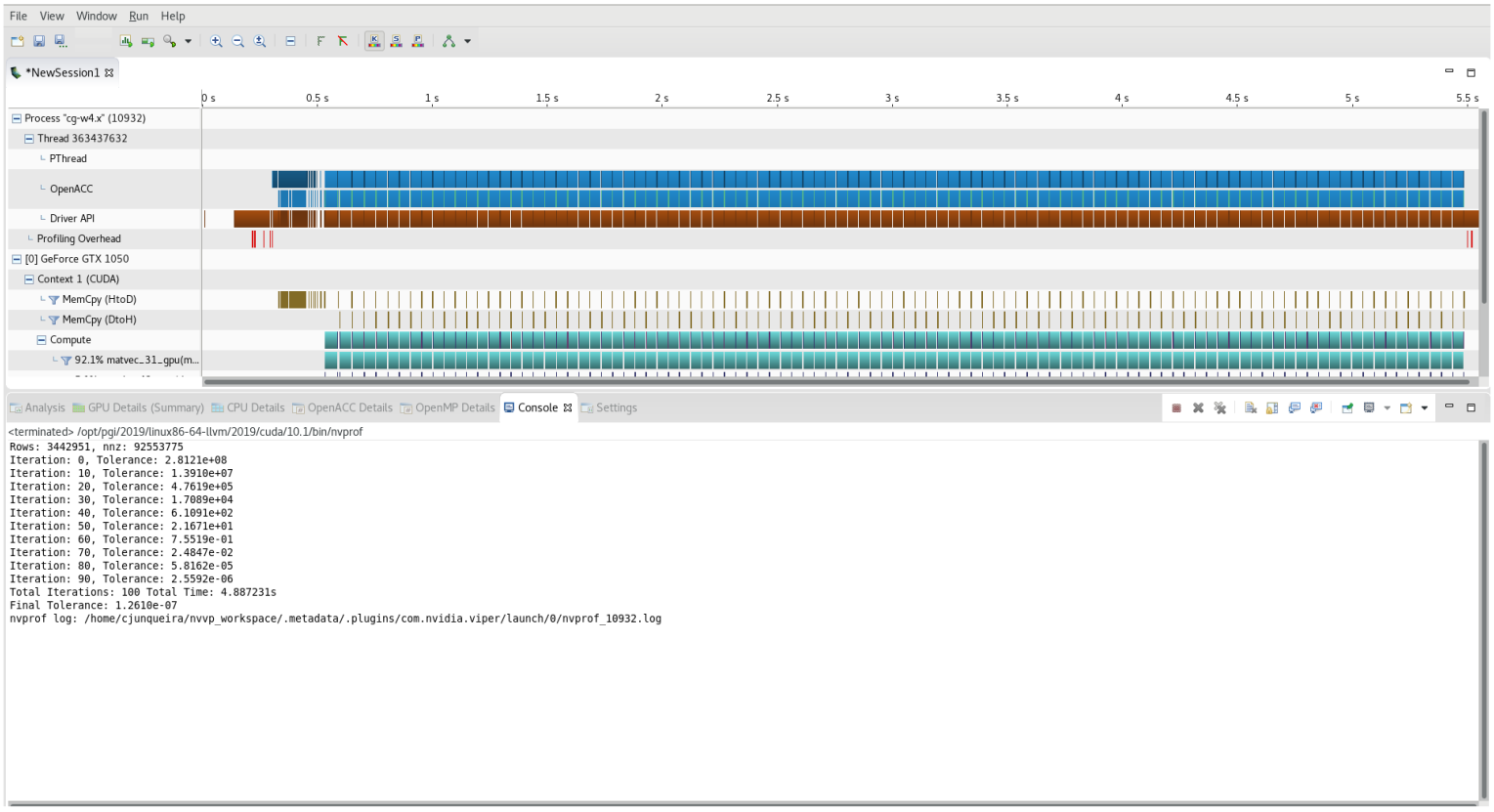
Increase Parallelism

PGProf Guided Analysis Mode - Nb. Workers = 32



Increase Parallelism

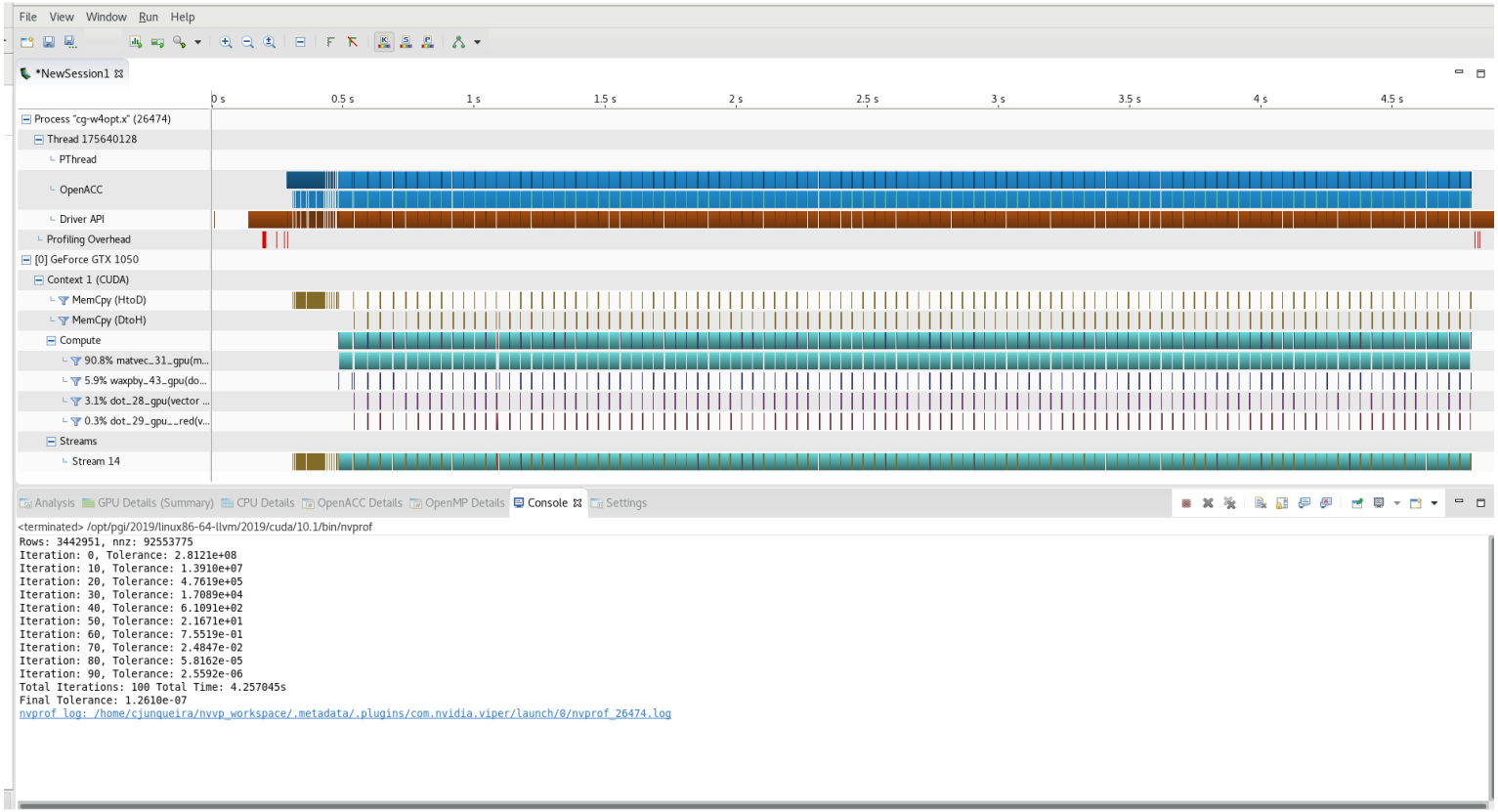
PGProf - Nb. Workers = 4



Increase Parallelism

PGProf - Nb. Workers = 4 - Compiled with fast

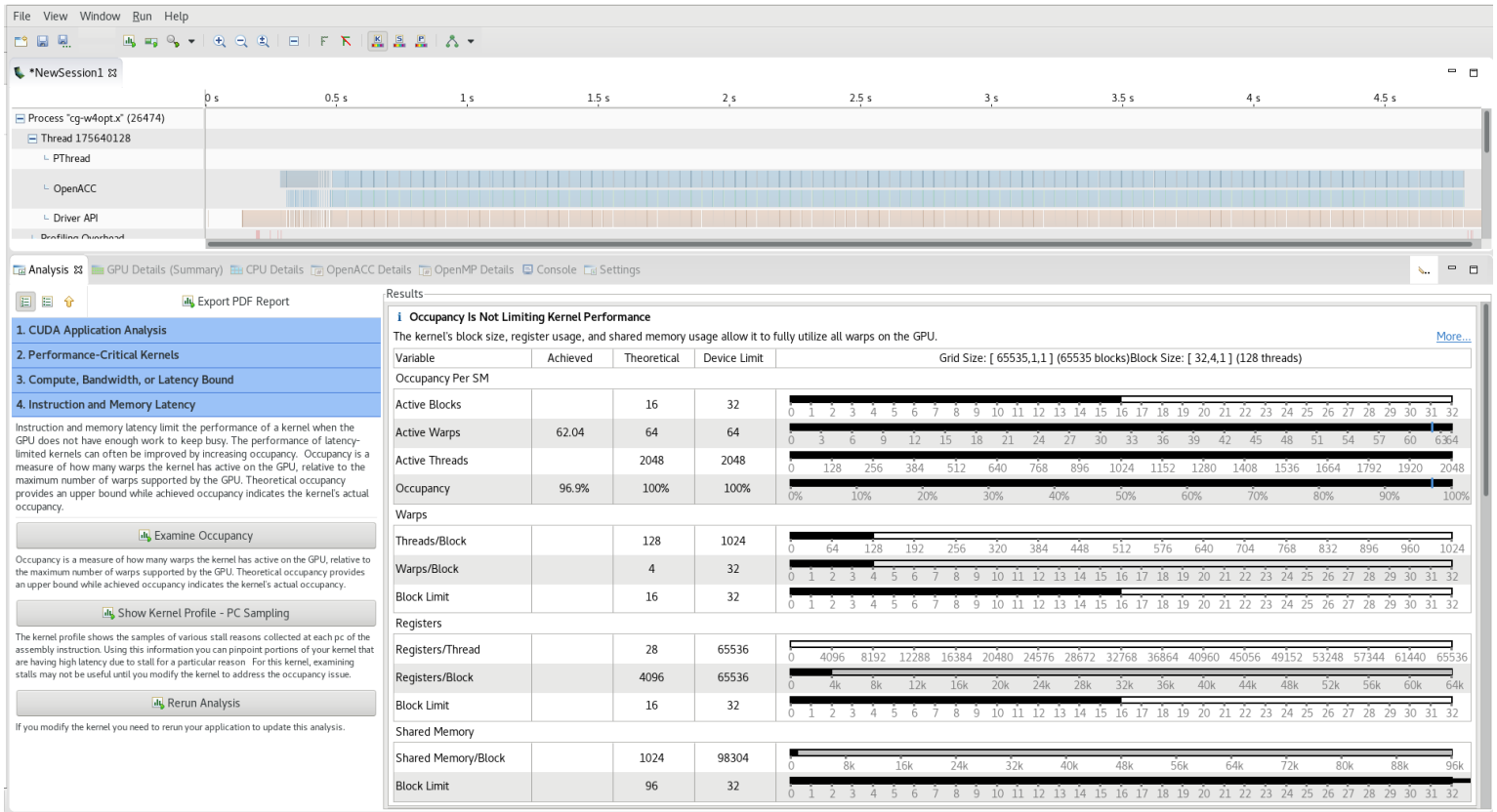
- Speedup fast: 7.14;



Increase Parallelism

PGProf - Nb. Workers = 4 - Compiled with fast

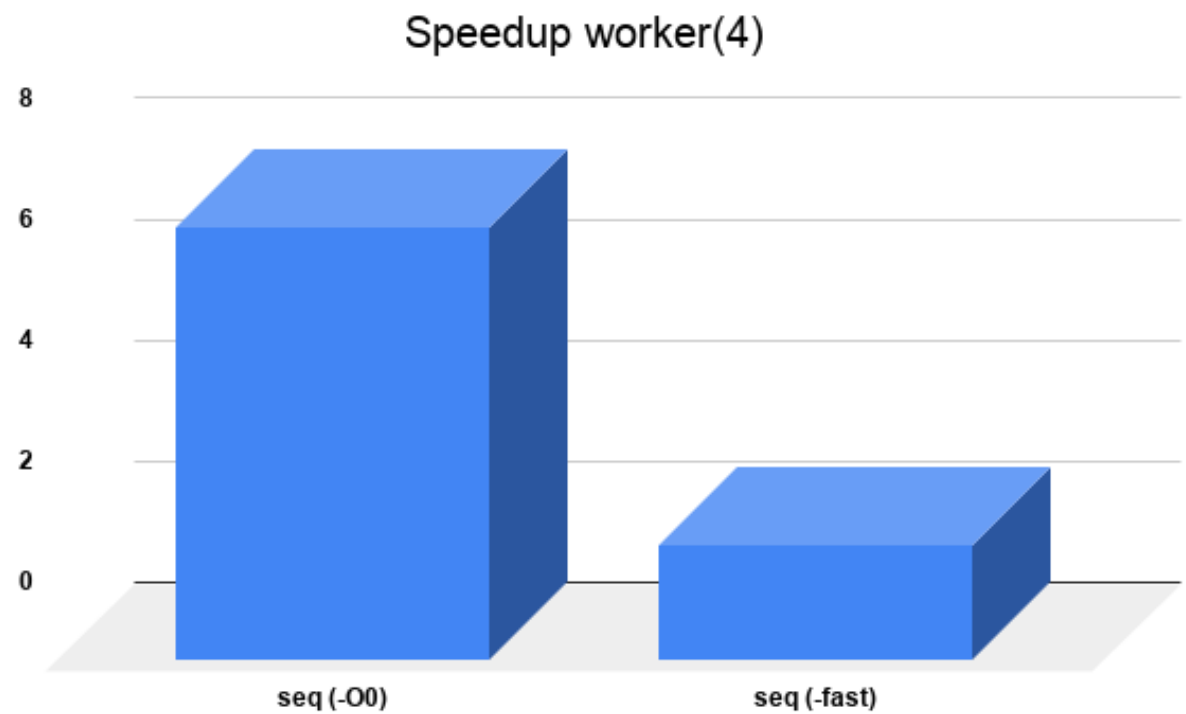
- Speedup fast: 7.14;



Increase Parallelism

PGProf - Nb. Workers = 4 - Compiled with fast

- Speedup fast/O0: 7.14;
- Speedup fast/fast: 1.9;



OpenACC Multicore

Running Parallel in Multicore

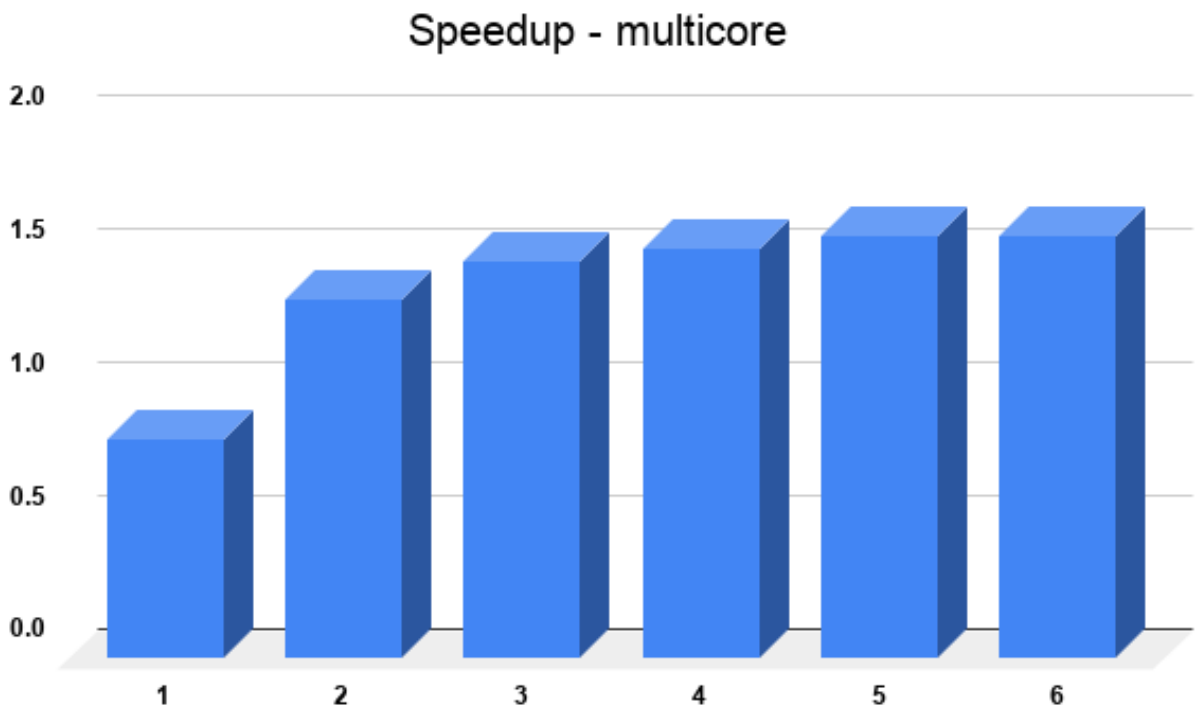
- OpenACC is not a GPU programming model, but rather a general model for parallel programming;
- The description of parallelism and data motion are applicable to any parallel architecture;
- The PGI compiler used in the presentation supports many architectures such as NVIDIA GPUs, AMD GPUs and multicore x86 CPUs;
- One can choose the x86 architecture in the compilation by using **-ta=multicore**. Moreover, it is possible to adjust the number of CPU cores for the calculation by setting the *ACC_NUM_CORES* environment variable to the number of CPU cores to use.

```
2 CXXFLAGS=-fast -ta=multicore -Minfo=all -Mneginfo
```

OpenACC Multicore

Speedup in Multicore

- The performance plateau beyond 4 CPU cores is due to the benchmark performance limited by the available CPU bandwidth.



OpenACC Multicore

Speedup in Multicore

- The performance plateau beyond 4 CPU cores is due to the benchmark performance limited by the available CPU bandwidth.

```
pgc++ -fast -ta=multicore -Minfo=all -Mneginfo main.cpp -o cg.x
initialize_vector(vector &, double):
     20, include "vector.h"
          40, Memory set idiom, loop replaced by call to __c_mset8
dot(const vector &, const vector &):
     21, include "vector_functions.h"
          28, Loop is parallelizable
              Generating Multicore code
              28, #pragma acc loop gang
          28, Loop not vectorized/parallelized: not countable
              FMA (fused multiply-add) instruction(s) generated
          29, Generating implicit reduction(+:sum)
waxpby(double, const vector &, double, const vector &, const vector &):
     21, include "vector_functions.h"
          43, Loop is parallelizable
              Generating Multicore code
              43, #pragma acc loop gang
          43, Loop not vectorized/parallelized: not countable
              FMA (fused multiply-add) instruction(s) generated
```

```
allocate_3d_poisson_matrix(matrix &, int):
     22, include "matrix.h"
          43, Loop not fused: different loop trip count
          44, Loop not vectorized/parallelized: loop count too small
          59, Loop not vectorized: data dependency
matvec(const matrix &, const vector &, const vector &):
     23, include "matrix_functions.h"
          31, Loop is parallelizable
              Generating Multicore code
              31, #pragma acc loop gang
          31, Loop not vectorized/parallelized: not countable
              FMA (fused multiply-add) instruction(s) generated
          36, Loop is parallelizable
              Loop not vectorized: non-stride-1 array reference
              Loop not vectorized: mixed data types
              Loop unrolled 2 times
```

OpenACC Summary

Summary

1. Incrementally describe the available parallelism to the compiler. When performing this step on architectures with distinct host and accelerator memories, it is common for the code to slow down during this step;
2. Describe data motion of the application. Compilers must always be cautious with data movement to ensure correctness, but developers are able to see the bigger picture and understand how data is shared between OpenACC regions in different functions. Performance will improve significantly on architectures with distinc memories after describing the data and data motion to the compiler;
3. Finally, it is necessary to use the knowledge of the application and the target architecture to optimize the loops. Frequently loop optimizations will provide only small performance gains, but it is sometimes possible to give the compiler more information about the loop than it would see otherwise to obtain even larger performance gains.