2D Poisson Equation solution using by Finite Difference Method using simple Decomposition of Mesh.

Fahad Tanveer

61702

Contents

[Introduction 3](#_Toc532147347)

[CUDA Implementation in COLAB 6](#_Toc532147348)

[Results 13](#_Toc532147349)

[CUDA Analysis 14](#_Toc532147350)

# Introduction

The Poisson equation is a 2D (elliptical) PDE with many applications. Poisson's equation is

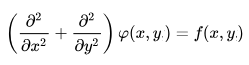


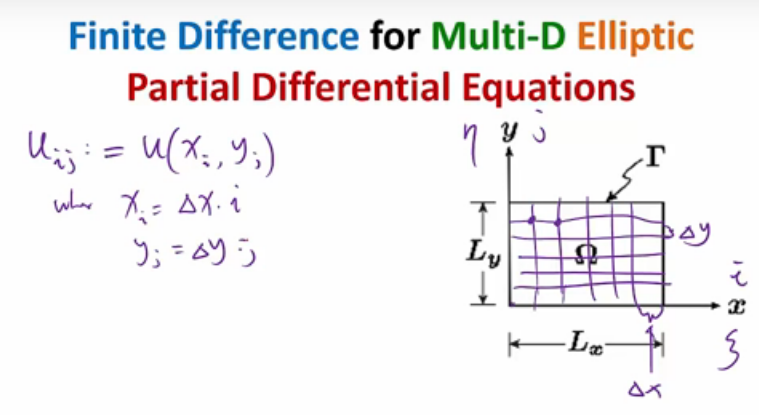
where {\displaystyle \Delta }Δ is the Laplace operator, and Φ{\displaystyle f} and {\displaystyle \varphi }f are real or complex-valued functions on a manifold.

When the manifold is Euclidean space, the Laplace operator is often denoted as ∇2 and so Poisson's equation is frequently written as

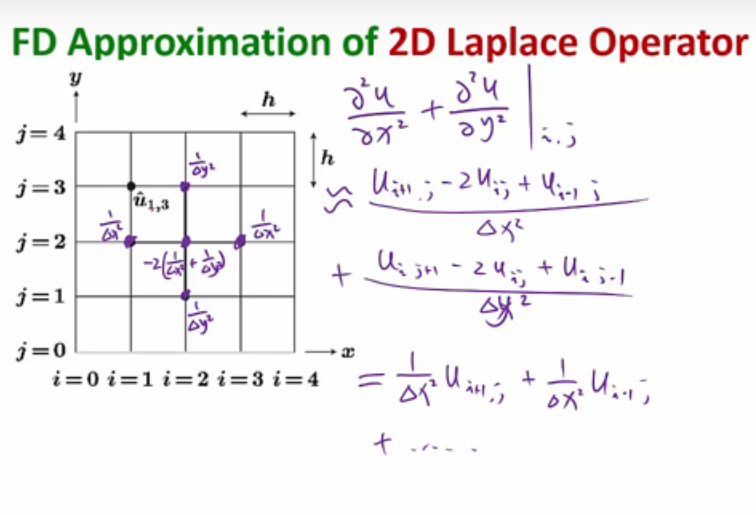


In 2D Cartesian coordinates, it takes the form

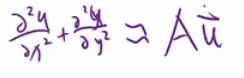




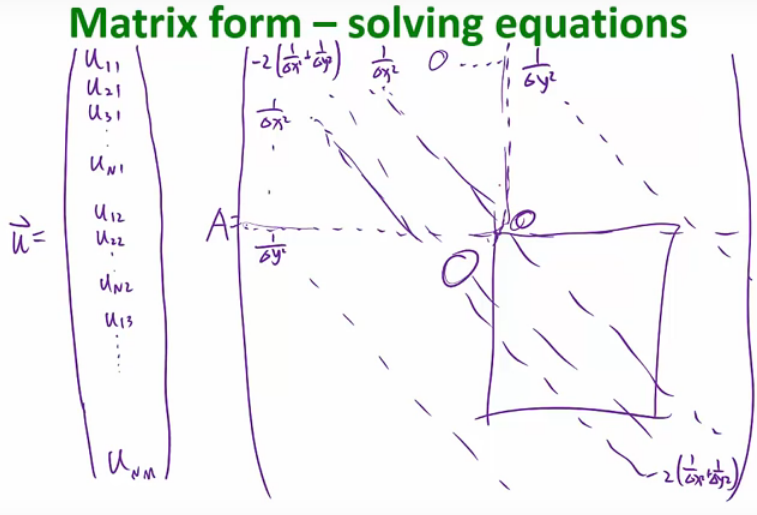
We use a 5 point stencil for our finite difference operator, to approximate laplace operator as shown in figure below:



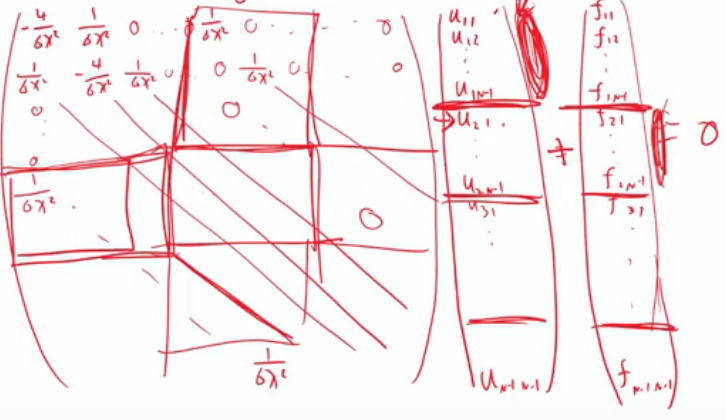
Finally we can convert it to matrix form



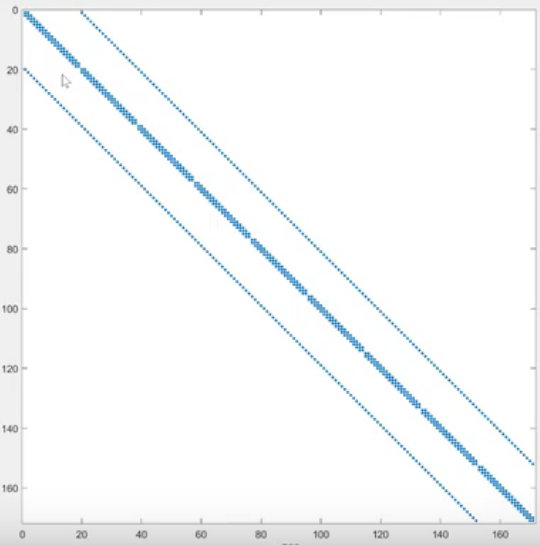
For flattened 2D input data set we have following matirces



The solution of this matrix can be divided into independent blocks for computation. So that we can make use of GPUs for computation.



The structure of the matrix is some what of the form as follows, we can observe in MATLAB using spy(A).



# CUDA Implementation in COLAB

We have assigned all oundary conditions =1

And hardcoded the code for a 10 x 10 input dataset with 10 iterations.

The CUDA code in colab is as follows:

|  |
| --- |
| %%cu  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Objective : Program to solve a solution of Poisson Eq. (PDE) on GPU    Input : No. of Grid Points in X-Dir, No. of Grid Points in Y-Dir  and maximum number of iterations    Output : Solution Vector.  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include<stdio.h>  #include<cuda.h>  #include<stdlib.h>  #include<sys/time.h>  //---------------------------------------------------------------------------  #define BLOCKSIZE 16  #define TOLERANCE 1.0E-06  //#define TOPBOUNDARYVALUE 4.1f  //#define BOTTOMBOUNDARYVALUE 3.1f  //#define LEFTBOUNDARYVALUE 1.1f  //#define RIGHTBOUNDARYVALUE 2.1f  #define TOPBOUNDARYVALUE 1.0f  #define BOTTOMBOUNDARYVALUE 1.0f  #define LEFTBOUNDARYVALUE 1.0f  #define RIGHTBOUNDARYVALUE 1.0f  //----------------------------------------------------------------------------  void IntializeAndSetBoundaryConditions(float \*\*, float \*\*, int , int , int );  void SetBoundaryCondition(int , int , float , int , float \*, float \*);  void IntializeUInteriorIndex(int \*\*, int , int , int );  float GetTheMaximumValue(float \*,int );  void IntializeUDifference(float \*\*, int );  //------------------------------------------------------------------------  //Pragma routine to report the detail of cuda error  #define CUDA\_SAFE\_CALL(call) \  do{ \  cudaError\_t err = call; \  if(err != cudaSuccess) \  { \  fprintf(stderr, "Cuda error in file '%s' in line %i : %s.\n", \  \_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString( err) ); \  exit(1); \  } \  } while (0) \  //------------------------------------------------------------------------------------------  //Kernel that performs the Jacobi Iteration  \_\_global\_\_ void JacobiIteration(float \*DeviceUOld, float \*DeviceUNew, int \*DeviceUInteriorIndex, float \*DeviceUDifference, int NoPointsX, int Size, int ThreadDim)  {  int tidx = threadIdx.x;  int tidy = threadIdx.y;  int ThreadIndex = (ThreadDim \* tidx) + tidy;  int MaxNumThread = ThreadDim \* ThreadDim;  int CurrentColumnIndex;  int pass = 0;  int Center, Left, Right, Bottom, Top;  while( (CurrentColumnIndex = (ThreadIndex + MaxNumThread \* pass)) < Size )  {  Center = DeviceUInteriorIndex[CurrentColumnIndex];  Left = Center - 1;  Right = Center + 1;  Top = Center - NoPointsX;  Bottom = Center + NoPointsX;  //Updating the UNew values  DeviceUNew[Center] = 0.25 \* (DeviceUOld[Left] + DeviceUOld[Right] + DeviceUOld[Top] + DeviceUOld[Bottom]);    //Finding the Difference between UNew and UOld  DeviceUDifference[CurrentColumnIndex] = DeviceUNew[Center] - DeviceUOld[Center];  //Assigning UNew to UOld  DeviceUOld[Center] = DeviceUNew[Center];  pass++;  }  \_\_syncthreads();  }//End of Jacobi Iteration Device function  //----------------------------------------------------------------------------------------------  int main(int argc, char \*\*argv)  {  //Checking if valid number of Arguements have been passed  /\*  if(argc != 4)  {  printf("Valid number of inputs are not given \n");  printf("Usage:<./Program Name><Number of X points><Number of Y points><Maximum Number of Iterations> \n");  exit(-1);  }  \*/  //Host Variables Declaration  float \*UOld, \*UNew, \*UDifference;  int \*UInteriorIndex;  float MaxError = 0.0f;  struct timeval TV;  double StartTime,EndTime,ActualTime;  int NoPointsX, NoPointsY, MaxIterations, NoPointsInterior, Index, PresentIteration,NoPointsTotal;  //Device Variables Declaration  float \*DeviceUOld, \*DeviceUNew, \*DeviceUDifference;  int \*DeviceUInteriorIndex;  //Obtaining the Values of NoPointsX, NoPointsY and MaxIterations from the arguements passed by the User  // NoPointsX = atoi( argv[1] );  //NoPointsY = atoi( argv[2] );  //MaxIterations = atoi( argv[3] );  NoPointsX = 10;  NoPointsY = 10;  MaxIterations = 10;  //Calculating the Total Points and Interior Points  NoPointsTotal = NoPointsX \* NoPointsY;  NoPointsInterior = (NoPointsTotal) - (((2 \* NoPointsX) + (2 \* NoPointsY)) - 4);  //Intializing the UOld and seting the Boundary conditions  IntializeAndSetBoundaryConditions( &UOld, &UNew, NoPointsX, NoPointsY, NoPointsTotal );  //Intializing the UDifference  IntializeUDifference( &UDifference,NoPointsInterior );  //Filling the UInteriorIndex with Index Values of Interior Points  IntializeUInteriorIndex( &UInteriorIndex, NoPointsX, NoPointsY,NoPointsInterior );    //Allocating Memory on Device  CUDA\_SAFE\_CALL( cudaMalloc( (void \*\*)&DeviceUOld, NoPointsTotal \* sizeof(float)));  CUDA\_SAFE\_CALL( cudaMalloc( (void \*\*)&DeviceUNew, NoPointsTotal \* sizeof(float)));  CUDA\_SAFE\_CALL( cudaMalloc( (void \*\*)&DeviceUInteriorIndex, NoPointsInterior \* sizeof(int)));  CUDA\_SAFE\_CALL( cudaMalloc( (void \*\*)&DeviceUDifference, NoPointsInterior \* sizeof(float)));  //Copying Data from Host to Device  CUDA\_SAFE\_CALL( cudaMemcpy((void \*)DeviceUOld, (void \*)UOld, NoPointsTotal \* sizeof(float), cudaMemcpyHostToDevice) );  CUDA\_SAFE\_CALL( cudaMemcpy((void \*)DeviceUNew, (void \*)UNew, NoPointsTotal \* sizeof(float), cudaMemcpyHostToDevice) );  CUDA\_SAFE\_CALL( cudaMemcpy((void \*)DeviceUInteriorIndex, (void \*)UInteriorIndex, NoPointsInterior \* sizeof(float), cudaMemcpyHostToDevice) );  CUDA\_SAFE\_CALL( cudaMemcpy((void \*)DeviceUDifference, (void \*)UDifference, NoPointsInterior \* sizeof(float), cudaMemcpyHostToDevice) );  //Defining Thread Grid and the Thread Block  dim3 DimGrid( 1,1 );  dim3 DimBlock( BLOCKSIZE,BLOCKSIZE );  PresentIteration = 0;  //start timing computation  gettimeofday(&TV, NULL);  StartTime = TV.tv\_sec+( TV.tv\_usec/1000000.0 );  while(1)  {  //Incrementing the Iteration Number  PresentIteration++;  //Invoking the Kernel  JacobiIteration<<<DimGrid, DimBlock>>>( DeviceUOld, DeviceUNew, DeviceUInteriorIndex, DeviceUDifference, NoPointsX, NoPointsInterior, BLOCKSIZE );    //Copying Udifference from Device to Host  CUDA\_SAFE\_CALL( cudaMemcpy((void \*)UDifference, (void \*)DeviceUDifference, NoPointsInterior \* sizeof(float), cudaMemcpyDeviceToHost) );  //Finding the Maximum among the UDifference values  MaxError = GetTheMaximumValue( UDifference, NoPointsInterior );  //Checking for the convergence  if((MaxError < TOLERANCE) || (PresentIteration == MaxIterations))  break;  }  //stop timing computation  gettimeofday(&TV,NULL);  EndTime = TV.tv\_sec+(TV.tv\_usec/1000000.0);    //calculate difference between start and stop times  ActualTime = EndTime - StartTime;  //Copying UNew from Device to Host  CUDA\_SAFE\_CALL(cudaMemcpy((void \*)UNew, (void \*)DeviceUNew, NoPointsTotal \* sizeof(float), cudaMemcpyDeviceToHost));    //Printing the solution  for(Index = 0; Index < NoPointsTotal; Index++)  printf(" %f", UNew[Index]);  printf("Output Vector given above calculated in %d Iterations and in %lf secs.\n",PresentIteration,ActualTime);  //Freeing the Allocated Memory on Device  CUDA\_SAFE\_CALL( cudaFree( DeviceUOld ) );  CUDA\_SAFE\_CALL( cudaFree( DeviceUNew ) );  CUDA\_SAFE\_CALL( cudaFree( DeviceUInteriorIndex ) );  CUDA\_SAFE\_CALL( cudaFree( DeviceUDifference ) );  //Freeing the Allocated Memory on Host  free( UOld );  free( UNew );  free( UInteriorIndex );  free( UDifference );  return(0);  }//End of Main  //-----------------------------------------------------------------------------------------------------  void IntializeAndSetBoundaryConditions( float \*\*UOld, float \*\*UNew, int NoPointsX, int NoPointsY, int NoPointsTotal )  {  float \*TempUOld,\*TempUNew;  int Index;  //Allocating memory for UOld and UNew  TempUOld = (float \*)malloc( NoPointsTotal \* sizeof(float) );  if(TempUOld == NULL)  {  printf("Can't allocate the memory for the variable TempUOld \n");  exit(-1);  }  TempUNew = (float \*)malloc( NoPointsTotal \* sizeof(float) );  if(TempUNew == NULL)  {  printf("Can't allocate the memory for the variable TempUNew \n");  exit(-1);  }    //Intialize UOld to zeros  for(Index = 0; Index < (NoPointsTotal); Index++)  TempUOld[Index] = 0.0;  //Setting the Boundary Conditions  //Case:Left  for(Index = 0; Index < NoPointsY; Index++)  SetBoundaryCondition(0, Index, LEFTBOUNDARYVALUE, NoPointsX, TempUOld, TempUNew);  //Case:Right  for(Index = 0; Index < NoPointsY; Index++)  SetBoundaryCondition((NoPointsX - 1), Index, RIGHTBOUNDARYVALUE, NoPointsX, TempUOld, TempUNew);  //Case:Bottom  for(Index = 0; Index < NoPointsX; Index++)  SetBoundaryCondition(Index, 0, BOTTOMBOUNDARYVALUE, NoPointsX, TempUOld, TempUNew);  //Case:Top  for(Index = 0; Index < NoPointsX; Index++)  SetBoundaryCondition(Index, (NoPointsY - 1), TOPBOUNDARYVALUE, NoPointsX, TempUOld, TempUNew);  //Assigning Temporary Varibles Locations to Original Variables  \*UOld = TempUOld;  \*UNew = TempUNew;  }  //---------------------------------------------------------------------------------------------------  void SetBoundaryCondition(int i, int j, float Value, int NoPointsX, float \*UOld, float \*UNew)  {  int Index;  Index = (j \* NoPointsX) + i;  UOld[Index] = Value;  UNew[Index] = Value;  }  //------------------------------------------------------------------------------------------------  void IntializeUInteriorIndex(int \*\*UInteriorIndex, int NoPointsX, int NoPointsY,int NoPointsInterior)  {  int i, j, Index, IndexValue;  int \*TempUInteriorIndex;  Index = 0;  //Allocating memory for UInteriorIndex  TempUInteriorIndex = (int \*)malloc( NoPointsInterior \* sizeof(int) );  if( TempUInteriorIndex == NULL )  {  printf("Can't allocate memory for the variable TempUInteriorIndex \n");  exit(-1);  }  //Assigning the index of the Interior points of UOld and UNew  for(j = 1; j < (NoPointsY - 1); ++j)  {  for(i = 1; i < (NoPointsX - 1); i++)  {  IndexValue = (j \* NoPointsX) + i;  TempUInteriorIndex[Index] = IndexValue;  Index++;  }  }    \*UInteriorIndex = TempUInteriorIndex;  }  //--------------------------------------------------------------------------------------------------------  float GetTheMaximumValue(float \*Array,int NumberOfElements)  {  float MaxError;  int RowNum;  MaxError = 0.0f;  for(RowNum = 0; RowNum < NumberOfElements; RowNum++)  {  if(Array[RowNum] >= MaxError)  MaxError = Array[RowNum];  }  return(MaxError);  }  //---------------------------------------------------------------------------------------------------------------  void IntializeUDifference(float \*\*UDifference, int NoPointsInterior)  {  float \*TempUDifference;  int RowNumber;    //Allocating Memory for UDifference  TempUDifference = (float \*)malloc( NoPointsInterior \* sizeof(float) );  if( TempUDifference == NULL )  {  printf("Can't allocate the memory for the variable TempUDifference \n");  exit(-1);  }  //Intializing the UDifference to zero's  for(RowNumber = 0; RowNumber < NoPointsInterior; RowNumber++)  TempUDifference[RowNumber] = 0.0f;  \*UDifference = TempUDifference;  }  //------------------------------------------------------------------------------------------------------- |

# Results

The code generates an output stream

|  |
| --- |
| ' 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000Output Vector given above calculated in 10 Iterations and in 0.000394 secs.\n' |

The code executes on 0.000394 seconds and outputs the 100 coefficients of u.

We can plot them in MATLAB using following commads:

|  |
| --- |
| a=[ 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000];  b=reshape(a,10,10);  surf(b); |

# 

# CUDA Analysis

We use following statements in COLAB to determine device properties

|  |
| --- |
| %cd /usr/local/cuda/samples/1\_Utilities/deviceQuery/  !ls  !make  !./deviceQuery |

The Code is running on a Tesla K80 GPU with following specifications:

|  |
| --- |
| /usr/local/cuda-9.2/samples/1\_Utilities/deviceQuery  deviceQuery.cpp Makefile NsightEclipse.xml readme.txt  /usr/local/cuda-9.2/bin/nvcc -ccbin g++ -I../../common/inc -m64 -gencode arch=compute\_30,code=sm\_30 -gencode arch=compute\_35,code=sm\_35 -gencode arch=compute\_37,code=sm\_37 -gencode arch=compute\_50,code=sm\_50 -gencode arch=compute\_52,code=sm\_52 -gencode arch=compute\_60,code=sm\_60 -gencode arch=compute\_61,code=sm\_61 -gencode arch=compute\_70,code=sm\_70 -gencode arch=compute\_70,code=compute\_70 -o deviceQuery.o -c deviceQuery.cpp  /usr/local/cuda-9.2/bin/nvcc -ccbin g++ -m64 -gencode arch=compute\_30,code=sm\_30 -gencode arch=compute\_35,code=sm\_35 -gencode arch=compute\_37,code=sm\_37 -gencode arch=compute\_50,code=sm\_50 -gencode arch=compute\_52,code=sm\_52 -gencode arch=compute\_60,code=sm\_60 -gencode arch=compute\_61,code=sm\_61 -gencode arch=compute\_70,code=sm\_70 -gencode arch=compute\_70,code=compute\_70 -o deviceQuery deviceQuery.o  mkdir -p ../../bin/x86\_64/linux/release  cp deviceQuery ../../bin/x86\_64/linux/release  ./deviceQuery Starting...  CUDA Device Query (Runtime API) version (CUDART static linking)  Detected 1 CUDA Capable device(s)  Device 0: "Tesla K80"  CUDA Driver Version / Runtime Version 9.2 / 9.2  CUDA Capability Major/Minor version number: 3.7  Total amount of global memory: 11441 MBytes (11996954624 bytes)  (13) Multiprocessors, (192) CUDA Cores/MP: 2496 CUDA Cores  GPU Max Clock rate: 824 MHz (0.82 GHz)  Memory Clock rate: 2505 Mhz  Memory Bus Width: 384-bit  L2 Cache Size: 1572864 bytes  Maximum Texture Dimension Size (x,y,z) 1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)  Maximum Layered 1D Texture Size, (num) layers 1D=(16384), 2048 layers  Maximum Layered 2D Texture Size, (num) layers 2D=(16384, 16384), 2048 layers  Total amount of constant memory: 65536 bytes  Total amount of shared memory per block: 49152 bytes  Total number of registers available per block: 65536  Warp size: 32  Maximum number of threads per multiprocessor: 2048  Maximum number of threads per block: 1024  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)  Max dimension size of a grid size (x,y,z): (2147483647, 65535, 65535)  Maximum memory pitch: 2147483647 bytes  Texture alignment: 512 bytes  Concurrent copy and kernel execution: Yes with 2 copy engine(s)  Run time limit on kernels: No  Integrated GPU sharing Host Memory: No  Support host page-locked memory mapping: Yes  Alignment requirement for Surfaces: Yes  Device has ECC support: Enabled  Device supports Unified Addressing (UVA): Yes  Device supports Compute Preemption: No  Supports Cooperative Kernel Launch: No  Supports MultiDevice Co-op Kernel Launch: No  Device PCI Domain ID / Bus ID / location ID: 0 / 0 / 4  Compute Mode:  < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >  deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.2, CUDA Runtime Version = 9.2, NumDevs = 1  Result = PASS |

We run the profiling tool nvprof which gives following results:

|  |
| --- |
| !ls  !nvcc Poisson2D.cu -o Poisson2D  !./Poisson2D |

|  |
| --- |
| !nvprof ./Poisson2D |

|  |
| --- |
| ==5581== NVPROF is profiling process 5581, command: ./Poisson2D  1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.702891 0.454813 0.286485 0.203747 0.203747 0.286485 0.454813 0.702891 1.000000 1.000000 0.733576 0.512134 0.360170 0.286485 0.286485 0.360170 0.512134 0.733576 1.000000 1.000000 0.797262 0.626961 0.512134 0.454813 0.454813 0.512134 0.626961 0.797262 1.000000 1.000000 0.889200 0.797262 0.733576 0.702891 0.702891 0.733576 0.797262 0.889200 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000 1.000000Output Vector given above calculated in 10 Iterations and in 0.000723 secs.  ==5581== Profiling application: ./Poisson2D  ==5581== Profiling result:  Type Time(%) Time Calls Avg Min Max Name  GPU activities: 56.57% 42.015us 10 4.2010us 4.1280us 4.7680us JacobiIteration(float\*, float\*, int\*, float\*, int, int, int)  33.61% 24.960us 11 2.2690us 2.1440us 2.7200us [CUDA memcpy DtoH]  9.82% 7.2960us 4 1.8240us 1.6000us 2.4000us [CUDA memcpy HtoD]  API calls: 99.10% 198.09ms 4 49.522ms 14.254us 198.00ms cudaMalloc  0.28% 560.33us 96 5.8360us 1.4130us 182.02us cuDeviceGetAttribute  0.23% 466.74us 15 31.116us 14.856us 44.033us cudaMemcpy  0.17% 339.07us 10 33.906us 17.079us 144.78us cudaLaunchKernel  0.10% 192.62us 4 48.154us 13.218us 140.40us cudaFree  0.09% 182.02us 1 182.02us 182.02us 182.02us cuDeviceTotalMem  0.02% 39.471us 1 39.471us 39.471us 39.471us cuDeviceGetName  0.00% 5.8040us 3 1.9340us 1.4050us 2.3900us cuDeviceGetCount  0.00% 4.5420us 1 4.5420us 4.5420us 4.5420us cuDeviceGetPCIBusId  0.00% 3.8660us 2 1.9330us 1.5270us 2.3390us cuDeviceGet |