An Introduction to CUDA using IPT

1. Introduction to IPT

The Interactive Parallelization Tool (IPT) is a tool for semi-automatically parallelizing existing serial programs written in C/C++. It supports parallelization using the three most popular parallel programming paradigms: OpenMP, MPI, and CUDA. The users interactively specify which regions or loops to parallelize in a serial program. On the basis of the specifications provided by the users, IPT can parallelize several types of C/C++ applications. It therefore frees the users from the burden of learning the low-level syntax of the different parallel programming paradigms, and in manually reengineering their serial code for creating parallel versions. In order to do so, IPT walks the users through a series of questions related to the parallelization of the input C/C++ programs and generates parallelized versions as output.

2. Introduction to GPU Programming

Graphic Processing Units (GPUs) are computing devices primarily designed for processing graphic and video inputs, and relaying the output to a display. GPUs are therefore designed to handle high throughput, where throughput is the number of tasks completed per unit of time. In comparison to a Central Processing Unit (CPU) that could have fewer but faster cores, a GPU could be having hundreds of cores that can handle thousands of threads. Therefore, even though a GPU's clock is slower than that of a CPU, the GPU is able to execute a greater number of commands per clock-cycle.

GPUs can be used for general-purpose programming and are used for running parallelized applications in several areas, such as physical simulations or machine learning. Nvidia and AMD are the two popular GPU manufacturers.

Compute Unified Device Architecture (CUDA) is a paradigm for parallelizing C and C++ programs by utilizing a system's GPU for general computing. In the early 2000's, the CUDA Toolkit was developed by Nvidia to provide programmers with a library and compiler for parallel programming on Nvidia GPUs.

In the CUDA programming model there is a **host,** and one or more **devices**. The CPU is the host, and the GPU is the device. The GPU acts as an accelerator to the CPU, which is the main processor or the host. The host and the device have separate memory. Therefore, the programmers may need to allocate memory on both the host and device, and explicitly write code to copy or move the data between them as needed. Similarly, one needs to demarcate the functions that run on the device from those running on the host. A function that runs on the device, and is called from the host, is called a **kernel**. The GPU responds to the kernels launched by the host (or the CPU). Only the host can launch kernels. Kernels are labeled with the identifier `__global__` to indicate to the CUDA compiler that this function will be called by the CPU to run on the GPU. In addition to global functions, there are also device functions that are demarcated with the identifier `__device__`**.** These functions can only be called from kernels or other device functions. The host cannot call device functions. Similarly, the host also has functions that can only be run by the host. For example the `main` function is always run from the host. Device and host functions are split at the compile-time. Host functions are processed by the standard compiler (such as `gcc`), and the device functions are processed by the `nvcc` compiler from the CUDA toolkit.

When a kernel is launched, the instructions are executed in a **thread**. The GPU can utilize thousands of threads to complete a task. Each thread will execute the same set of instructions from the kernel. Threads are run in **warps** which are groups of 32 threads that execute simultaneously. Each thread has its own local memory that is private to the thread, as well as an identification number. There is a three-dimension arragement of threads on a GPU: x, y, and z are the dimensions. This dimensionality can be utilized when parallelizing problems with multiple dimensions such as performing calculations on a 2-D matrix.

A group of threads that share memory is called a **block**. Threads running in the same block can access the same address space in memory. Depending on the GPU, the maximum number of threads in a block is either 512 or 1024. Blocks are assigned to run on **stream multiprocessors**, or **SMs** for short. SMs are hardware constructs that have registers, cache, and memory. SMs also have 2 warp schedulers. When a SM executes the threads in a block, it runs them in groups called warps. As mentioned before, warps are groups of up 32 threads that run simultaneously. If a block contains 64 threads, the SM will run them in two warps. If a block has 100 threads, it will run 4 warps. The first 3 warps will all contain 32 threads, and the last warp would have 4 threads and 28 dummy threads.

Blocks are themselves organized in a three-dimensional structure called a grid. Blocks also extend in the x, y, and z dimensions (this kernel dimensionality is referred to by variables with `dim3` type and will be covered in detail at a later point) and have their own identification number. When launching a kernel, the programmer specifies the number of blocks to run in each dimension, and also specified the number of threads in each block per dimension. There is no way to specify what SM a block is assigned to, or the order that the SM will run the blocks in. As a result, blocks must be independent from one another and must be able to be run in any order.

Indexing is another aspect of CUDA grid dimensionality. Often times kernels will be written based on the thread index. As both threads and blocks have identification numbers in the x, y, and z dimensions, CUDA has the following built-in variables related to identification and dimensions: `uint3 threadIdx, uint3 blockIdx, dim3 blockDim, dim3 gridDim`. The `uint3` data type is a coordinate variable in CUDA based on the typical `int` data type. These variables can be used to find either aggregate properties about programs (for example, how many kernels are running) or as indices to control thread actions. For example, in order to specify that a kernel only runs on threads in the z dimension, one can add an if statement that checks the thread is a z-dimension thread.

3. CUDA Program Structure

Many CUDA programs follow the same general structure since there is a defined relationship between the host and device. The CPU is responsible for most of the overhead for running a kernel such as moving data from the host to the device, launching kernels, and copying the results from the device back to the host. To launch a kernel, the CPU will allocate storage on the GPU before copying data from the CPU to the GPU. In order to allocate memory and copy the data, often the functions `cudaMalloc` (for allocating memory on the device), `cudaMemcpy` (for copying data from the host to the device, or vice versa), and `cudaFree` (for releasing used memory on the device) are used.

Executing a kernel is a concurrent operation in CUDA unless specified, therefore the CPU will continue executing instructions while the kernel is running. However, a new kernel operation cannot start until the previous one has concluded.

Figure 1 shows the general structure of a CUDA program, in which only one kernel is called. A kernel is launched using the syntax shown in Figure 2.

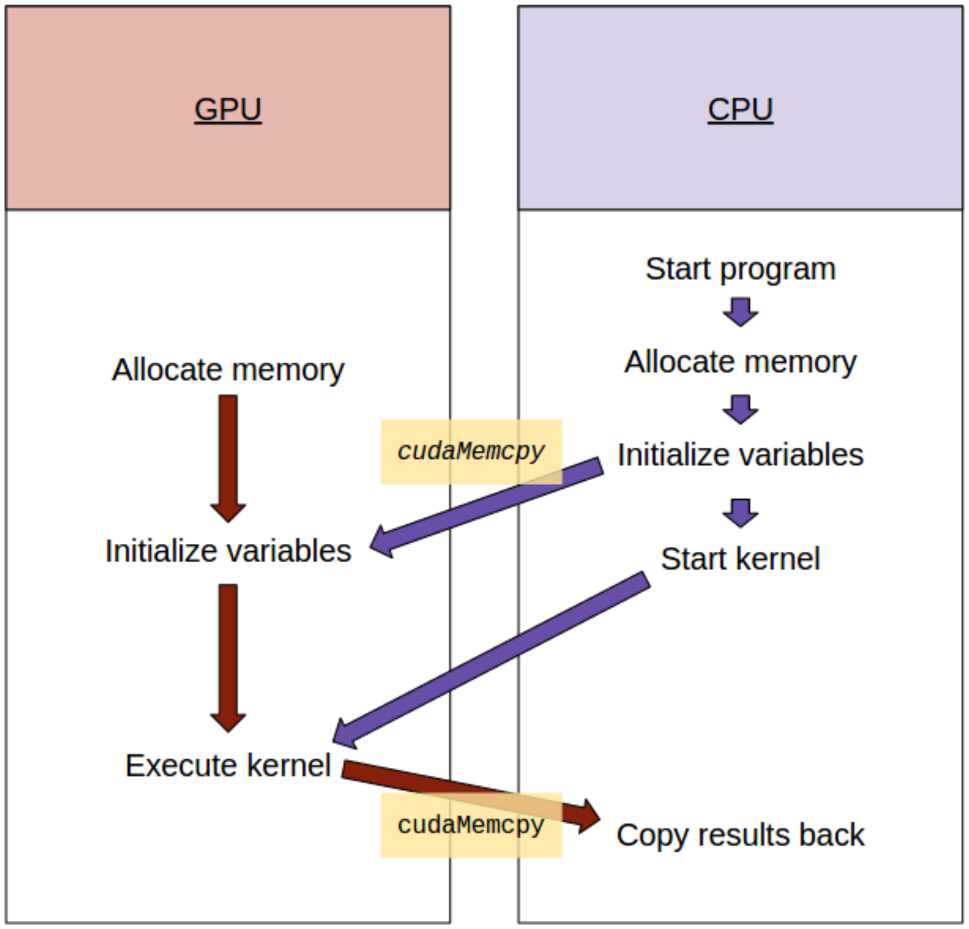

Figure 1: General Strcuture of a CUDA program

```
dim3 gridSize(M,1,1);

dim3 blockSize(N,1,1);

kernelName<<< gridSize, blockSize >>>(kernel Parameters);
```

Figure 2: A C code snippet showing the syntax for kernel launch

The three pairs of chevrons shown in Figure 2 are used to set the execution parameters. In Figure 2, the gridSize represents the number of blocks and the blockSize represents the number of threads in each block.

The parameters inside the chevrons are of **`dim3`** type and are useful for setting the dimensionality for kernel calls. In Figure 2, `dim3` was initialized with all three parameters but the user could choose to provide only 1 or 2 parameters depending on the problem being solved. Any parameter that is not initialized defaults to 1. To determine the total number of threads that a kernel is using, one must consider the number of threads in each dimension as well as the number of blocks in each dimension. The total number of threads is equal to the product of the number of blocks and the number of threads per block. To determine the number of threads in a block, multiply the number of threads in each dimension together for the thread `dim3` object. In the example code snippet in Figure 2, the total number of threads in each block is N*1*1 = N. The same process is repeated to find the total number of blocks in the grid M*1*1=M. The total number of threads used here is: M*N.

In summary, the program structure shown in Figure 1, should have the components listed in Figure 3. Note that, as mentioned before, because the GPU and CPU have separate memory, the GPU results obtained in the kernel must be returned to the CPU for further processing. This is done via `cudaMemcpy()` as well. Finally, the allocated GPU memory can be freed via `cudaFree()`.

1. CPU allocates storage on GPU (`cudaMalloc()`)
2. CPU copies data from CPU to GPU (`cudaMemcpy()`)
3. CPU launches kernel on GPU to perform some computation
4. CPU copies results back from GPU (`cudaMemcpy()`)

Figure 3: General components of the CUDA program represented in Figure 1

4. Generating the First CUDA Program with IPT

The C program shown in figure 4 can be used to compute the sum of all integers in the interval [0, 10000]. We will use this program as a test case for demonstarting the process of using IPT for generating CUDA code. The steps for parallelizing the code with IPT are shown in Figure 5. The Generated code is shown in Figure 6. The user may need to modify the number of threads and blocks that are specified in the generated code.

```
#include <stdio.h>
int main(int argc, char *argv[]){
   int i, sum,upToVal;
   upToVal = 10001;
   sum = 0;
   for(i=0; i< upToVal; i++){
      sum = sum +i;
   }
   printf("\nSum of first %d numbers is: %d\n\n",upToVal, sum);
   return 0;
}
```

Figure 4: C program to compute the sum of integers in the interval [0, 10000]

```
Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
3

Please enter the function in which you wish to insert the kernel call(or
parallelize the for-loop).
Please choose the function that you want to parallelize from the list below
1 : main
1

Would you like
1. For-loop
2. TBD
1

for (i = 0; i < upToVal; i++) {
  sum = sum + i;
}
Is this the for loop you are looking for?(y/n)
y
Do you want to perform reduction on any variables ? (Y/N)
y

Please enter the variables to reduce ([ format:  1,2,3 etc. ] with 1 is for
the first variable, 2 is for the second variable and so on).
 Possible variables to reduce are:
1. sum type is int
2. i type is int
1

Please enter the reduction operation for variable [sum]. Possible reduction
operations are:
1. Sum
2. Product
1
Do you want to perform reduction on any arrays ? (Y/N)
n
```

Figure 5: IPT prompts and user responses (bold)

```
#include <stdio.h>
void __global__ kernel0(int sum[],int upToVal, int device_M , int device_N){
int i =  blockIdx.x * blockDim.x + threadIdx.x;
int  print_statement_deleted_here=0;
sum[i] = 0;
{
sum[i] = sum[i] + i;
}
__syncthreads();
__syncthreads();
}

int main(int argc,char *argv[])
{
  int total_sum = 0;
  int *host_sum;
  int *device_sum;
  int i;
  int sum;
  int upToVal;
  upToVal = 10001;
  sum = 0;
//Inserting code for memory allocation grid size and block size calculation
host_sum= (int*)malloc((1)*((int) ((upToVal - 0 ) / (1)))*sizeof(int));
cudaMalloc((void **) &device_sum,( 1)*((int) ((upToVal - 0 ) /
(1)))*sizeof(int));
//Please note this is the section wherein the number of blocks and threads
are calculated.  To change the number of threads alter the dimBlock whereas
to change the number of blocks alter the dimGrid
int D_rows = 1 + ( ((int) ((upToVal - 0 ) / (1)) > 1024 ) ? (int) ((upToVal
- 0 ) / (1))/1024 : (int) ((upToVal - 0 ) / (1)) );
int D_cols = ((int) ((upToVal - 0 ) / (1)) > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);
kernel0<<<dimGrid,dimBlock>>>(device_sum,upToVal,1,(int) ((upToVal - 0 ) /
(1)));
/*
  int IPT_function_replace;
*/
//Copying from Device to Host
cudaMemcpy(host_sum,device_sum,((int) ((upToVal - 0 ) / (1)))*sizeof(int),
cudaMemcpyDeviceToHost);
 //code for variable reduction
 for(long row = 0; row < 1; ++row){for(long col = 0; col < (int) ((upToVal -
0 ) / (1)); ++col)      {  total_sum+= host_sum[row*(int) ((upToVal - 0 ) /
(1))+ col]; }  }
sum+= total_sum;
//  Ending Parallelization
  printf("\nSum of first %d numbers is: %d\n\n",upToVal,sum);
  return 0;
}
```

Figure 6: CUDA code generated by IPT – sum of numbers

5. Parallelizing the Code for Finding Sum of Arrays

The program shown in Figure 7 computes the sum of two arrays (A, B) and saves the results in array C. The steps to parallelize this code with IPT are shown in Figure 8, and the generated code is shown in Figure 9.

```
#include <stdio.h>
int main() {
  int lm = 10000;
  int A[lm];
  int B[lm];
  int C[lm];
  for (int j=0; j < lm; j++) {
      A[j] = j;
      B[j] = j;
  }
  // Addition
  for (int q=0; q < lm; q++) {
      C[q] = A[q] + B[q];
  }
  // Checker
  for (int s=0; s < lm; s++) {
      if (C[s] != 2*s) {
      printf("failed at row %d\n", s);
      }
  }
}
```

Figure 7: Serial C program to compute the sum of arrays A and B

```
Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
3

Please enter the function in which you wish to insert the kernel call(or
parallelize the for-loop).
Please choose the function that you want to parallelize from the list below
1 : main
1

Would you like
1. For-loop
2. TBD
1

for (int j = 0; j < lm; j++) {
  A[j] = j;
  B[j] = j;
}
Is this the for loop you are looking for?(y/n)
```

```
n

OK - will find the next loop if available.

// Addition
for (int q = 0; q < lm; q++) {
  C[q] = A[q] + B[q];
}
Is this the for loop you are looking for?(y/n)
y
cannot automatically find the number of iteration for the loop, please
specify the number of iteration:
10001
Do you want to perform reduction on any variables ? (Y/N)
n
Do you want to perform reduction on any arrays ? (Y/N)
n
Is the following array [ C ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
2
Is the following array [ A ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
Is the following array [ B ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
```

Figure 8: IPT prompts and user responses (bold)

```
#include <stdio.h>
__global__  void kernel0(int * C,int * A,int * B,int lm, int device_M , int
device_N){
int q =  blockIdx.x * blockDim.x + threadIdx.x;
int  print_statement_deleted_here=0;
{
C[q] = A[q] + B[q];
}
__syncthreads();
}

int main()
{
  int *device_B;
  int *device_A;
  int *device_C;
  int lm = 10000;
  int A[lm];
  int B[lm];
  int C[lm];
  for (int j = 0; j < lm; j++) {
      A[j] = j;
      B[j] = j;
  }
// Addition
//Please note this is the section wherein the number of blocks and threads
are calculated.  To change the number of threads alter the dimBlock whereas
to change the number of blocks alter the dimGrid
int D_rows = 1 + ( (10001 > 1024 ) ? 10001/1024 : 10001 );
int D_cols = (10001 > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);
cudaMalloc((void **) &device_C,(lm)*sizeof(int));
cudaMalloc((void **) &device_A,(lm)*sizeof(int));
cudaMemcpy(device_A,A,(lm)*sizeof(int),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_B,(lm)*sizeof(int));
cudaMemcpy(device_B,B,(lm)*sizeof(int),cudaMemcpyHostToDevice);
kernel0<<<dimGrid,dimBlock>>>(device_C,device_A,device_B,lm,1,10001);
/*
  int IPT_function_replace;
*/
// Checker
cudaMemcpy(C,device_C,(lm)*sizeof(int), cudaMemcpyDeviceToHost);
cudaFree(device_C);
cudaFree(device_A);
cudaFree(device_B);
  for (int s = 0; s < lm; s++) {
      if (C[s] != 2 * s) {
      printf("failed at row %d\n",s);
      }
  }
}
```
Figure 9: CUDA code generated by IPT – sum of arrays

6. Parallelizing the Circuit Satisfiability Code

The program shown in Figure 10 is the serial version of the circuit satisfiability problem (https://people.sc.fsu.edu/~jburkardt/c_src/satisfy/satisfy.html).

This code simulated a logical ciruit can computes the number of combinations of 0s and 1s that are provided to the circuit for which the circuit produce an output of 1. The steps to parallelize this code with IPT are shown in Figure 11, and the generated code is shown in Figure 12. Note that only the number of solutions found is printed by the CUDA version.

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>

int main ( int argc, char *argv[] );
int circuit_value ( int n, int bvec[] );
void i4_to_bvec ( int i4, int n, int bvec[] );
void timestamp ( void );

/****************************************************************************
***/
```

```
int main ( int argc, char *argv[] )

/**************************************************************************
***/
/*
  Purpose:

      MAIN is the main program for SATISFY.

  Licensing:

      This code is distributed under the GNU LGPL license.

  Modified:

      20 March 2009

  Author:

      John Burkardt

  Reference:

      Michael Quinn,
      Parallel Programming in C with MPI and OpenMP,
      McGraw-Hill, 2004,
      ISBN13: 978-0071232654,
      LC: QA76.73.C15.Q55.
*/
{
# define N 23

  int bvec[N];
  int i;
  int ihi;
  int j;
  int n = N;
  int solution_num;
  int value;

  printf ( "\n" );
  timestamp ( );
  printf ( "\n" );
  printf ( "SATISFY\n" );
  printf ( "  C version\n" );
  printf ( "  We have a logical function of N logical arguments.\n" );
  printf ( "  We do an exhaustive search of all 2^N possibilities,\n" );
  printf ( "  seeking those inputs that make the function TRUE.\n" );
/*
  Compute the number of binary vectors to check.
*/
  ihi = 1;
  for ( i = 1; i <= n; i++ )
  {
      ihi = ihi * 2;
  }
  for ( i = 1; i <= n/2; i++ )
  {
      printf("\ntest\n");
  }
  printf ( "\n" );
  printf ( "  The number of logical variables is N = %d\n", n );
  printf ( "  The number of input vectors to check is %d\n", ihi );
  printf ( "\n" );
  printf ( "   #        Index ---------Input Values-----------------------
\n" );
  printf ( "\n" );
/*
  Check every possible input vector.
*/
  solution_num = 0;

  for ( i = 0; i < ihi; i++ )
  {
      i4_to_bvec ( i, n, bvec );

      value = circuit_value ( n, bvec );

      if ( value == 1 )
      {
      solution_num = solution_num + 1;

      printf ( "  %2d  %10d:  ", solution_num, i );
      for ( j = 0; j < n; j++ )
      {
      printf ( " %d", bvec[j] );
      }
      printf ( "\n" );
      }
  }
  //added this for testing. If the already parallelized loop is not the last
one, then there is no error, else segementation fault
  //for ( i = 1; i <= n/2; i++ )
  //{
  //  printf("\ntestagain\n");
  //}

//  Report.

  printf ( "\n" );
  printf ( "  Number of solutions found was %d\n", solution_num );
/*
  Shut down.
*/
  printf ( "\n" );
  printf ( "SATISFY\n" );
  printf ( "  Normal end of execution.\n" );
  printf ( "\n" );
  timestamp ( );

  return 0;
```

```
# undef N
}
/***************************************************************************
***/

int circuit_value ( int n, int bvec[] )

/***************************************************************************
***/
/*
  Purpose:

      CIRCUIT_VALUE returns the value of a circuit for a given input set.

  Licensing:

      This code is distributed under the GNU LGPL license.

  Modified:

      20 March 2009

  Author:

      John Burkardt

  Reference:

      Michael Quinn,
      Parallel Programming in C with MPI and OpenMP,
      McGraw-Hill, 2004,
      ISBN13: 978-0071232654,
      LC: QA76.73.C15.Q55.

  Parameters:

      Input, int N, the length of the input vector.

      Input, int BVEC[N], the binary inputs.

      Output, int CIRCUIT_VALUE, the output of the circuit.
*/
{
  int value;

  value =
       (  bvec[0]  ||  bvec[1]  )
    && ( !bvec[1]  || !bvec[3]  )
    && (  bvec[2]  ||  bvec[3]  )
    && ( !bvec[3]  || !bvec[4]  )
    && (  bvec[4]  || !bvec[5]  )
    && (  bvec[5]  || !bvec[6]  )
    && (  bvec[5]  ||  bvec[6]  )
    && (  bvec[6]  || !bvec[15] )
    && (  bvec[7]  || !bvec[8]  )
    && ( !bvec[7]  || !bvec[13] )
```

```
      && (  bvec[8]  ||  bvec[9]  )
      && (  bvec[8]  || !bvec[9]  )
      && ( !bvec[9]  || !bvec[10] )
      && (  bvec[9]  ||  bvec[11] )
      && (  bvec[10] ||  bvec[11] )
      && (  bvec[12] ||  bvec[13] )
      && (  bvec[13] || !bvec[14] )
      && (  bvec[14] ||  bvec[15] )
      && (  bvec[14] ||  bvec[16] )
      && (  bvec[17] ||  bvec[1]  )
      && (  bvec[18] || !bvec[0]  )
      && (  bvec[19] ||  bvec[1]  )
      && (  bvec[19] || !bvec[18] )
      && ( !bvec[19] || !bvec[9]  )
      && (  bvec[0]  ||  bvec[17] )
      && ( !bvec[1]  ||  bvec[20] )
      && ( !bvec[21] ||  bvec[20] )
      && ( !bvec[22] ||  bvec[20] )
      && ( !bvec[21] || !bvec[20] )
      && (  bvec[22] || !bvec[20] );

  return value;
}
/****************************************************************************
***/

void i4_to_bvec ( int i4, int n, int bvec[] )

/****************************************************************************
***/
/*
  Purpose:

      I4_TO_BVEC converts an integer into a binary vector.

  Licensing:

      This code is distributed under the GNU LGPL license.

  Modified:

      20 March 2009

  Author:

      John Burkardt

  Parameters:

      Input, int I4, the integer.

      Input, int N, the dimension of the vector.

      Output, int BVEC[N], the vector of binary remainders.
*/
{
  int i;

  for ( i = n - 1; 0 <= i; i-- )
  {
      bvec[i] = i4 % 2;
      i4 = i4 / 2;
  }

  return;
}
/*****************************************************************************
***/

void timestamp ( void )

/*****************************************************************************
***/
/*
  Purpose:

      TIMESTAMP prints the current YMDHMS date as a time stamp.

  Example:

      31 May 2001 09:45:54 AM

  Licensing:

      This code is distributed under the GNU LGPL license.

  Modified:

      24 September 2003

  Author:

      John Burkardt

  Parameters:

      None
*/
{
# define TIME_SIZE 40

  static char time_buffer[TIME_SIZE];
  //const struct tm *tm;
  //size_t len;
  //time_t now;

  //now = time ( NULL );
  //tm = localtime ( &now );

  //len = strftime ( time_buffer, TIME_SIZE, "%d %B %Y %I:%M:%S %p", tm );

  //printf ( "%s\n", time_buffer );
  return;
# undef TIME_SIZE
}
```

Figure 10: Serial C program for the circuit-satisfiability problem

```
Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
3

Please enter the function in which you wish to insert the kernel call(or
parallelize the for-loop).
Please choose the function that you want to parallelize from the list below
1 : main
2 : circuit_value
3 : i4_to_bvec
4 : timestamp
1

Would you like
1. For-loop
2. TBD
1

for (i = 1; i <= n; i++) {
  ihi = ihi * 2;
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (i = 1; i <= n / 2; i++) {
  printf("\ntest\n");
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (i = 0; i < ihi; i++) {
  i4_to_bvec(i,n,bvec);
  value = circuit_value(n,bvec);
  if (value == 1) {
      solution_num = solution_num + 1;
      printf("  %2d  %10d:  ",solution_num,i);
      for (j = 0; j < n; j++) {
      printf(" %d",bvec[j]);
      }
      printf("\n");
  }
}
Is this the for loop you are looking for?(y/n)
y
Do you want to perform reduction on any variables ? (Y/N)
y

Please enter the variables to reduce ([ format:  1,2,3 etc. ] with 1 is for
the first variable, 2 is for the second variable and so on).
 Possible variables to reduce are:
1. i type is int
2. n type is int
3. value type is int
4. solution_num type is int
5. print_statement_deleted_here type is int
4

Please enter the reduction operation for variable [solution_num]. Possible
reduction operations are:
1. Sum
2. Product
1
Do you want to perform reduction on any arrays ? (Y/N)
n
Is the following array [ bvec ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
4
```

Figure 11: IPT prompts and user responses (bold)

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>
int main(int argc,char *argv[]);
int circuit_value(int n,int bvec[]);
void i4_to_bvec(int i4,int n,int bvec[]);
void timestamp();
/*****************************************************************************
***/
__device__ void i4_to_bvec_rose(int i4,int n,int bvec[]){
int i;
for(i = n - 1;0 <= i;i--) {bvec[i] = i4 % 2;i4 = i4 / 2;}
return ;
}
__device__ int circuit_value_rose(int n,int bvec[]){
int value;
value =(bvec[0] || bvec[1]) &&(!bvec[1] || !bvec[3]) &&(bvec[2] || bvec[3])
&&(!bvec[3] || !bvec[4]) &&(bvec[4] || !bvec[5]) &&(bvec[5] || !bvec[6])
&&(bvec[5] || bvec[6]) &&(bvec[6] || !bvec[15]) &&(bvec[7] || !bvec[8])
&&(!bvec[7] || !bvec[13]) &&(bvec[8] || bvec[9]) &&(bvec[8] || !bvec[9])
&&(!bvec[9] || !bvec[10]) &&(bvec[9] || bvec[11]) &&(bvec[10] || bvec[11])
```

```
&&(bvec[12] || bvec[13]) &&(bvec[13] || !bvec[14]) &&(bvec[14] || bvec[15])
&&(bvec[14] || bvec[16]) &&(bvec[17] || bvec[1]) &&(bvec[18] || !bvec[0])
&&(bvec[19] || bvec[1]) &&(bvec[19] || !bvec[18]) &&(!bvec[19] || !bvec[9])
&&(bvec[0] || bvec[17]) &&(!bvec[1] || bvec[20]) &&(!bvec[21] || bvec[20])
&&(!bvec[22] || bvec[20]) &&(!bvec[21] || !bvec[20]) &&(bvec[22] ||
!bvec[20]);
return value;
}
__global__  void kernel0(int solution_num[],int ihi,int n,int value,int j,
int device_M , int device_N){
int i =  blockIdx.x * blockDim.x + threadIdx.x;
int  print_statement_deleted_here=0;
int bvec[23];
solution_num[i] = 0;
{
i4_to_bvec_rose(i,n,bvec);
value = circuit_value_rose(n,bvec);
if(value == 1) {solution_num[i] = solution_num[i] +
1;print_statement_deleted_here;for(j = 0;j < n;j++)
{print_statement_deleted_here;}print_statement_deleted_here;}
}
__syncthreads();
__syncthreads();
}

int main(int argc,char *argv[])
/***********************************************************************
***/
/*
  Purpose:
      MAIN is the main program for SATISFY.
  Licensing:
      This code is distributed under the GNU LGPL license.
  Modified:
      20 March 2009
  Author:
      John Burkardt
  Reference:
      Michael Quinn,
      Parallel Programming in C with MPI and OpenMP,
      McGraw-Hill, 2004,
      ISBN13: 978-0071232654,
      LC: QA76.73.C15.Q55.
*/
{
  int total_solution_num = 0;
  int *host_solution_num;
  int *device_solution_num;
# define N 23
  int bvec[23];
  int i;
  int ihi;
  int j;
  int n = 23;
  int solution_num;
  int value;
  printf("\n");
  timestamp();
  printf("\n");
  printf("SATISFY\n");
  printf("  C version\n");
  printf("  We have a logical function of N logical arguments.\n");
  printf("  We do an exhaustive search of all 2^N possibilities,\n");
  printf("  seeking those inputs that make the function TRUE.\n");
/*
  Compute the number of binary vectors to check.
*/
  ihi = 1;
  for (i = 1; i <= n; i++) {
      ihi = ihi * 2;
  }
  for (i = 1; i <= n / 2; i++) {
      printf("\ntest\n");
  }
  printf("\n");
  printf("  The number of logical variables is N = %d\n",n);
  printf("  The number of input vectors to check is %d\n",ihi);
  printf("\n");
  printf("   #    Index ---------Input Values------------------------\n");
  printf("\n");
/*
  Check every possible input vector.
*/
  solution_num = 0;
//Inserting code for memory allocation grid size and block size calculation
host_solution_num= (int*)malloc((1)*((int) ((ihi - 0 ) / (1)))*sizeof(int));
cudaMalloc((void **) &device_solution_num,( 1)*((int) ((ihi - 0 ) /
(1)))*sizeof(int));
//Please note this is the section wherein the number of blocks and threads
are calculated.  To change the number of threads alter the dimBlock whereas
to change the number of blocks alter the dimGrid
int D_rows = ((int) ((ihi - 0 ) / (1)) > 1024 ) ? (int) ((ihi - 0 ) /
(1))/1024 : (int) ((ihi - 0 ) / (1));
int D_cols = ((int) ((ihi - 0 ) / (1)) > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);
kernel0<<<dimGrid,dimBlock>>>(device_solution_num,ihi,n,value,j,1,(int)
((ihi - 0 ) / (1)));
/*
  int IPT_function_replace;
*/
//added this for testing. If the already parallelized loop is not the last
one, then there is no error, else segementation fault
//for ( i = 1; i <= n/2; i++ )
//{
//  printf("\ntestagain\n");
//}
//  Report.
//Copying from Device to Host
cudaMemcpy(host_solution_num,device_solution_num,((int) ((ihi - 0 ) /
(1)))*sizeof(int), cudaMemcpyDeviceToHost);
 //code for variable reduction
 for(long row = 0; row < 1; ++row){for(long col = 0; col < (int) ((ihi - 0 )
/ (1)); ++col)    {  total_solution_num+= host_solution_num[row*(int) ((ihi
- 0 ) / (1))+ col];     }  }
solution_num+= total_solution_num;
//  Ending Parallelization
  printf("\n");
  printf("  Number of solutions found was %d\n",solution_num);
/*
  Shut down.
*/
  printf("\n");
  printf("SATISFY\n");
  printf("  Normal end of execution.\n");
  printf("\n");
  timestamp();
  return 0;
# undef N
}
/**************************************************************************
***/

int circuit_value(int n,int bvec[])
/**************************************************************************
***/
/*
  Purpose:
      CIRCUIT_VALUE returns the value of a circuit for a given input set.
  Licensing:
      This code is distributed under the GNU LGPL license.
  Modified:
      20 March 2009
  Author:
      John Burkardt
  Reference:
      Michael Quinn,
      Parallel Programming in C with MPI and OpenMP,
      McGraw-Hill, 2004,
      ISBN13: 978-0071232654,
      LC: QA76.73.C15.Q55.
  Parameters:
      Input, int N, the length of the input vector.
      Input, int BVEC[N], the binary inputs.
      Output, int CIRCUIT_VALUE, the output of the circuit.
*/
{
  int value;
  value = (bvec[0] || bvec[1]) && (!bvec[1] || !bvec[3]) && (bvec[2] ||
bvec[3]) && (!bvec[3] || !bvec[4]) && (bvec[4] || !bvec[5]) && (bvec[5] ||
!bvec[6]) && (bvec[5] || bvec[6]) && (bvec[6] || !bvec[15]) && (bvec[7] ||
!bvec[8]) && (!bvec[7] || !bvec[13]) && (bvec[8] || bvec[9]) && (bvec[8] ||
!bvec[9]) && (!bvec[9] || !bvec[10]) && (bvec[9] || bvec[11]) && (bvec[10]
|| bvec[11]) && (bvec[12] || bvec[13]) && (bvec[13] || !bvec[14]) &&
(bvec[14] || bvec[15]) && (bvec[14] || bvec[16]) && (bvec[17] || bvec[1]) &&
(bvec[18] || !bvec[0]) && (bvec[19] || bvec[1]) && (bvec[19] || !bvec[18])
&& (!bvec[19] || !bvec[9]) && (bvec[0] || bvec[17]) && (!bvec[1] ||
bvec[20]) && (!bvec[21] || bvec[20]) && (!bvec[22] || bvec[20]) &&
(!bvec[21] || !bvec[20]) && (bvec[22] || !bvec[20]);
  return value;
}
/*************************************************************************
***/

void i4_to_bvec(int i4,int n,int bvec[])
/*************************************************************************
***/
/*
  Purpose:
      I4_TO_BVEC converts an integer into a binary vector.
  Licensing:
      This code is distributed under the GNU LGPL license.
  Modified:
      20 March 2009
  Author:
      John Burkardt
  Parameters:
      Input, int I4, the integer.
      Input, int N, the dimension of the vector.
      Output, int BVEC[N], the vector of binary remainders.
*/
{
  int i;
  for (i = n - 1; 0 <= i; i--) {
      bvec[i] = i4 % 2;
      i4 = i4 / 2;
  }
  return ;
}
/*************************************************************************
***/

void timestamp()
/*************************************************************************
***/
/*
  Purpose:
      TIMESTAMP prints the current YMDHMS date as a time stamp.
  Example:
      31 May 2001 09:45:54 AM
  Licensing:
      This code is distributed under the GNU LGPL license.
  Modified:
      24 September 2003
  Author:
      John Burkardt
  Parameters:
      None
*/
{
# define TIME_SIZE 40
//  static char time_buffer[40];
//  const struct tm *tm;
//  size_t len;
//  time_t now;
//  now = time(((void *)0));
//  tm = (localtime((&now)));
//  len = strftime(time_buffer,40,"%d %B %Y %I:%M:%S %p",tm);
//  printf("%s\n",time_buffer);
  return ;
# undef TIME_SIZE
}
```
Figure 12: Generated CUDA code – circuit-staisfiability

7. Parallelizing Matrix Multiplication

The serial C++ program shown in Figure 13 calculates the product of two matrixes. The steps to parallelize this code are shown in Figire 14, and the generated CUDA code is shown in Figure 15. This code can be run as-is without any modifcations.

```
#include <iostream>
#include <stdio.h>
#include <cstdlib>

using namespace std;

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

  //int ** mult = (int**)malloc (sizeof(int)*M);
  //int *a = (int*) malloc (sizeof(int)*M);
  //int *b = (int*) malloc (sizeof(int)*K);
  const int M = 100;
  const int N = 100;
  const int K = 100;

  int mult[M][N];
  int a[M][K];
  int b[K][N];
  //a = &a_t;
  //b = &b_t;
  //mult = &mult_t;
  cout << "here\n";
  for (int i = 0; i < M; ++i) {
      for ( int k = 0; k < K; k++) {
         a[i][k] = i;
      }
  }
  for (int k = 0; k < K; ++k) {

      for ( int j = 0; j < N; j++) {
      b[k][j] = j;
      }
  }
  for (int i = 0; i < M; ++i) {
      for ( int j = 0; j < N; j++) {
      mult[i][j] = 0;
      }
  }

  for(int i = 0; i < M; ++i) {
      for(int j = 0; j < N; ++j ) {
            for(int k = 0; k < K; ++k)
            {
                  mult[i][j] += a[i][k] * b[k][j];
            }
    }
}
  for (int i = 0; i < M; ++i) {
      for ( int j = 0; j < N; j++) {
                  cout << mult[i][j] << " ";
      }
    cout << "\n";
  }
  return 0;

}
```

Figure 13: C++ program to compute the multiplication of matrices A and B

```
Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
3

Please enter the function in which you wish to insert the kernel call(or
parallelize the for-loop).
Please choose the function that you want to parallelize from the list below
1 : main
1

Would you like
1. For-loop
2. TBD
1
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int k = 0; k < K; k++) {
      a[i][k] = i;
  }
}
Is this the for loop you are looking for?(y/n)
n
```

```
OK - will find the next loop if available.

for (int k = 0; k < K; k++) {
  a[i][k] = i;
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (int k = 0; k < K; ++k) {
  for (int j = 0; j < N; j++) {
      b[k][j] = j;
  }
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (int j = 0; j < N; j++) {
  b[k][j] = j;
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int j = 0; j < N; j++) {
      mult[i][j] = 0;
  }
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.

for (int j = 0; j < N; j++) {
  mult[i][j] = 0;
}
Is this the for loop you are looking for?(y/n)
n

OK - will find the next loop if available.
```

```
Note: With your response, you will be selecting or declining the
parallelization of the outermost for-loop in the code region shown below. If
instead of the outermost for-loop, there are any inner for-loops in this
code region that you are interested in parallelizing, then, you will be able
to select those at a later stage.

for (int i = 0; i < M; ++i) {
  for (int j = 0; j < N; ++j) {
      for (int k = 0; k < K; ++k) {
      mult[i][j] += a[i][k] * b[k][j];
      }
  }
}
Is this the for loop you are looking for?(y/n)
y
cannot automatically find the number of iteration for the loop, please
specify the number of iteration:
100
Do you want to perform reduction on any variables ? (Y/N)
n
Do you want to perform reduction on any arrays ? (Y/N)
n
Is the following array [ mult ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
3
Is the following array [ a ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
Is the following array [ b ]
1. Input , 2. Output 3. Input/Output 4. Neither Input nor Output
1
```

Figure 14: IPT prompts and user responses (bold)

```
#include <iostream>
#include <stdio.h>
#include <cstdlib>
using namespace std;
__global__  void kernel0(int * mult,int * a,int * b,const int M,const int
N,const int K, int device_M , int device_N){
int i =  blockIdx.x * blockDim.x + threadIdx.x;
int  print_statement_deleted_here=0;
{
for(int j = 0;j < N;++j) {for(int k = 0;k < K;++k) {mult[j + i *(1 * 100)]
+= a[k + i *(1 * 100)] * b[j + k *(1 * 100)];}}
}
__syncthreads();
}

int main(int argc,char **argv)
{
  int *device_b;
  int *device_a;
  int *device_mult;
//int ** mult = (int**)malloc (sizeof(int)*M);
```

```
//int *a = (int*) malloc (sizeof(int)*M);
//int *b = (int*) malloc (sizeof(int)*K);
  const int M = 100;
  const int N = 100;
  const int K = 100;
  int mult[100][100];
  int a[100][100];
  int b[100][100];
//a = &a_t;
//b = &b_t;
//mult = &mult_t;
  cout<<"here\n";
  for (int i = 0; i < M; ++i) {
      for (int k = 0; k < K; k++) {
      a[i][k] = i;
      }
  }
  for (int k = 0; k < K; ++k) {
      for (int j = 0; j < N; j++) {
      b[k][j] = j;
      }
  }
  for (int i = 0; i < M; ++i) {
      for (int j = 0; j < N; j++) {
      mult[i][j] = 0;
      }
  }
//Please note this is the section wherein the number of blocks and threads
are calculated.  To change the number of threads alter the dimBlock whereas
to change the number of blocks alter the dimGrid
int D_rows = (100 > 1024 ) ? 100/1024 : 100;
int D_cols = (100 > 1024 ) ? 1024 : 1;
dim3 dimGrid(D_rows,1);
dim3 dimBlock(D_cols,1);
cudaMalloc((void **) &device_mult,(100)*(100)*sizeof(int));
for(int rose_i0 = 0; rose_i0 < 100;rose_i0++)
cudaMemcpy(device_mult + rose_i0*100,mult[
rose_i0],(100)*sizeof(int),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_a,(100)*(100)*sizeof(int));
for(int rose_i0 = 0; rose_i0 < 100;rose_i0++)
cudaMemcpy(device_a + rose_i0*100,a[
rose_i0],(100)*sizeof(int),cudaMemcpyHostToDevice);
cudaMalloc((void **) &device_b,(100)*(100)*sizeof(int));
for(int rose_i0 = 0; rose_i0 < 100;rose_i0++)
cudaMemcpy(device_b + rose_i0*100,b[
rose_i0],(100)*sizeof(int),cudaMemcpyHostToDevice);
kernel0<<<dimGrid,dimBlock>>>(device_mult,device_a,device_b,M,N,K,1,100);
/*
  int IPT_function_replace;
*/
for(int rose_i0 = 0; rose_i0 < 100;rose_i0++)
cudaMemcpy(mult[ rose_i0],device_mult + rose_i0*100,(100)*sizeof(int),
cudaMemcpyDeviceToHost);
cudaFree(device_mult);
cudaFree(device_a);
cudaFree(device_b);
```

```
  for (int i = 0; i < M; ++i) {
      for (int j = 0; j < N; j++) {
      (cout << mult[i][j])<<" ";
      }
      cout<<"\n";
  }
  return 0;
}
```

Figure 15: Generated CUDA Code

8. CUDA Code Optimizations

A key design point that will help program optimization is to write kernels that look like serial programs and can be run on only one thread. While it is computationally inexpensive to launch and run thousands of threads on a GPU, interdependencies between threads will slow-down a program. It is best to design the kernel like a generalized serial program that is run on a single thread.

Data transfer between the CPU and GPU is essential but this process can be costly in terms of time and resources. Therefore, it is important to minimize the data transfers between the CPU and GPU. This means that the ratio of computation to communication should be kept as high as possible. This can be achieved by ensuring that the computations performed by the GPU are not trivial and are worth running on the device.

A related factor to consider when making performance modifications is maximizing arithmetic density. Arithmetic density is the ratio of math:memory time. This ratio can be increased by maximizing work and computation operations per thread or minimizing memory time spent on memory access per thread. Time can be minimized on memory access using faster memory for frequently accessed data. Local memory is the fastest to access, followed by shared, with global being the slowest. Another way to decrease the time spent on memory access is to use coalesced memory for global memory access. Time used for memory access will be minimal when the threads read from and write to contiguous memory locations.

Similarly, when launching the kernels in a thread, it is also important to be mindful of thread divergence. Thread divergence occurs when some threads may execute a statement that others do not due to logical branching. Divergence is most often caused by if-statements and loops. Divergence hurts performance because of implicit barriers that prevent threads from moving on. Consider a group of 32 threads running in a warp with identification numbers from 0-31. If these threads are running on a kernel with a loop that runs as many times as the square thread’s identification number, then threads with lower identification number will finish much faster than the threads with higher identification numbers. The thread with identification number 0 will run through the code once, while thread with identification number 32 will run the code 1024 times. All threads in the warp will wait until the last thread finishes. While this example is rather extreme it demonstrates how thread divergence can impact the performance.

9. Common Operations

CUDA programs allow a few parallel communication models. These operations are gather, scatter, map, stencil, and transpose. Gather is a many-to-one operation where threads read multiple elements to produce a single output. Scatter is a one-to-many operation where an output is written to many memory locations. Map is a one-to-one pattern where read and write operations only occur at a specific location. Stencil is a specific type of gather operation where the output is dependent upon the neighboring data points. Transpose is a reordering tool to change how data is arranged in memory. It can be useful to simplify data structures for calculations such as flattening matrices into arrays.

10. Synchornization

Apart from communication between threads, CUDA also offers the possibility of synchronizing them. CUDA threads running in a kernel can be programmed to do so until reaching a barrier - called by `__syncthreads()`, where they are all forced to wait until all threads reach the barrier. Barriers help in preventing threads from reading from or writing to memory locations before the program is ready for them. However one must be wary of using too many synchronization barriers because it does slow-down the program since some threads will become idle in the waiting processes.

Similarly, there is an implicit barrier between two kernels. This means that if a program calls kernel A and then kernel B, the program will finish kernel A before moving on to kernel B. If the programmer wants to add in an explicit barrier, `__syncthreads()` can be used. Recall that threads are run in groups of 32 called warps. In order to synchronize the threads in a warp, one needs to call `__syncthreads()`. Note that `__syncthreads()` will only synchronize threads in the same block and does not synchronize at the grid or device level.

Another way to control threads is to use atomic operations. Atomic operations help prevent memory collisions where threads read and write from the same locations at the same time. When memory collisions occur, data can be corrupted resulting in incorrect results. Atomic operations serialize memory accesses so that only one thread can access for reading or writing at a time. While atomic actions are useful for preventing memory collisions, serialization in this manner will decrease program speed.