Journal of Signal Processing Systems (2019) 91:235–245
https://doi.org/10.1007/s11265-018-1395-2

GPUBlocks: GUI Programming Tool for CUDA and OpenCL

Yuan-Shin Hwang[1] · **Hsih-Hsin Lin**[1] · **Shen-Hung Pai**[1] · **Chia-Heng Tu**[2]

Received: 15 November 2016 / Revised: 5 February 2018 / Accepted: 9 July 2018 / Published online: 24 July 2018

Abstract
Recent advances in general-purpose graphics processing units (GPGPUs) have resulted in massively parallel hardware that is widely available to achieve high performance in desktop, notebook, and even mobile computer systems. While multicore technology has become the norm of modern computers, programming such systems requires the understanding of underlying hardware architecture and hence posts a great challenge for average programmers, who might be professionals in specific domains, but not experts in parallel programming. This paper presents a GUI tool called *GPUBlocks* that can facilitate parallel programming on multicore computer systems. GPUBlocks is developed based on the OpenBlocks framework, an extendable tool for graphical programming, to construct the GUI-based programming environment for CUDA and OpenCL parallel computing platforms. Programmers simply need to drag-n-drop blocks, fill the fields of the blocks, and connect them according to array or matrix computations that are specified by algorithms. GPUBlocks can then translate block-based code to CUDA or OpenCL programs. Furthermore, a couple of optimization constructs have also been offered for rapid program optimization. Experimental results have shown that the generated CUDA and OpenCL programs can achieve reasonable speedups on GPUs. Consequently, GPUBlocks can be used as a tool for fast prototyping of GPU applications or a platform for educational parallel programming.

Keywords GPGPU · CUDA · OpenCL · Heterogeneous computing · Programming tool · GUI

1 Introduction

Nowadays, multicore systems equipped with homogeneous or heterogeneous processing elements are popular and common in modern computers. For example, world leading semiconductor chip makers, such as Intel, IBM, AMD, MediaTek, and Qualcomm, add data accelerators, e.g., digital signal processor and graphical processing unit, to improve system performance for certain types of data computation jobs. There are lots of programming paradigms that have been proposed to facilitate programming for such heterogeneous platforms, such as OpenMP [6], OpenMPI [14], OpenCL [5], OpenVX [15], and CUDA [13].

While these programming facilities are helpful to abstract underlying hardware at certain level, they are simply too complicated for average programmers or algorithm developers. Therefore, a more intuitive environment can greatly reduce the burden of parallel programming. There have been several visual programming frameworks proposed to simplify general programming tasks, such as OpenBlocks [7], NetLogo [12], StarLogo The Next Generation (TNG) [11], Scratch [10], Blockly [3], and Scratch Blocks [16]. In addition, several tools have been designed to support specific programming languages, such as Android App Inventor for Android system [9], and Hopscotch for Apple iOS [4]. However, there are few tools available to ease the pain of parallel programming on multicore platforms.

This paper presents a graphical programming tool for CUDA and OpenCL called *GPUBlocks*. A prototype implementation of GPUBlocks has been constructed upon the open source visual programming frameworks OpenBlocks [7] and ArduBlock [1]. Major building blocks have been developed so that the CUDA and OpenCL programs can be generated automatically after users graphically specify array and matrix computations of target applications. Furthermore, several optimization blocks have also been developed to quickly optimize the program performance by switching from the standard code blocks to the optimized

✉ Yuan-Shin Hwang
shin@csie.ntust.edu.tw

[1] Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei 106, Taiwan

[2] Department of Computer Science and Information Engineering, National Cheng Kung University, Tainan 701, Taiwan

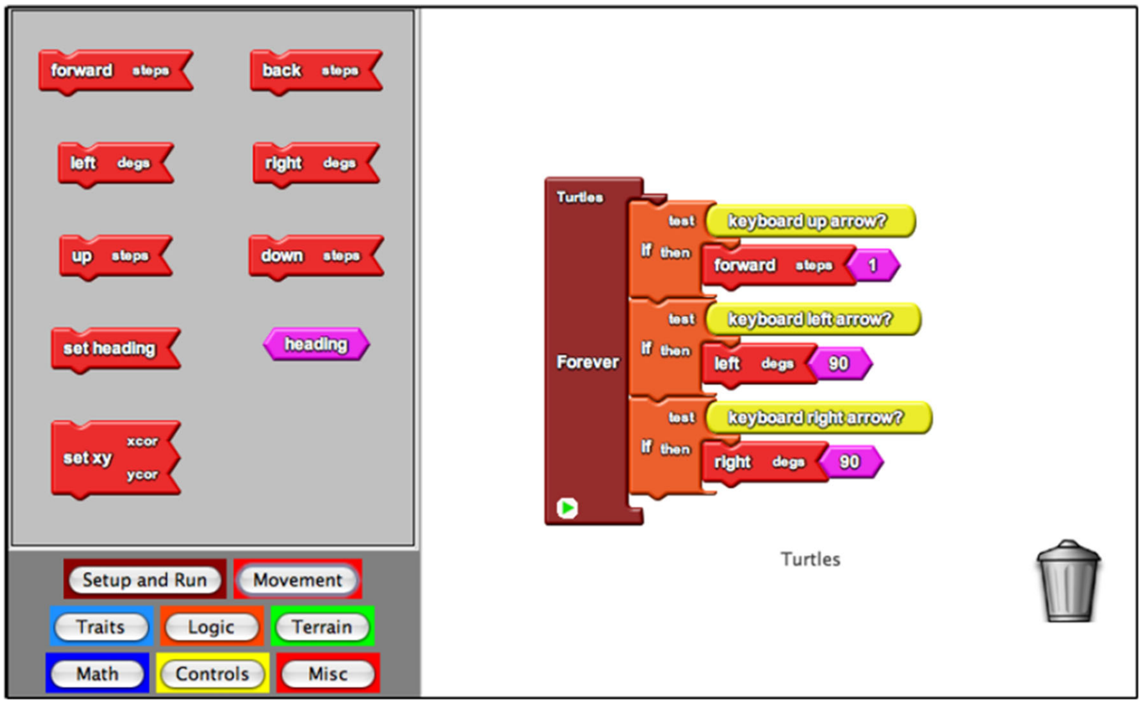


Figure 1 OpenBlocks.

ones. In addition, as the tool is capable of displaying the generated CUDA or OpenCL programs alongside corresponding CUDA or OpenCL blocks, beginners will be able to learn the CUDA or OpenCL programming by comparing the blocks against the generated codes. Experimental results have shown that the generated CUDA and OpenCL programs can achieve reasonable speedups on GPUs.

The main results of this paper are as follows:

- This paper presents a GUI tool called *GPUBlocks* that integrates the visual programming approach to facilitate parallel programming on GPUs.
- Programmers simply need to drag-n-drop blocks, fill the fields of the blocks, and connect them according

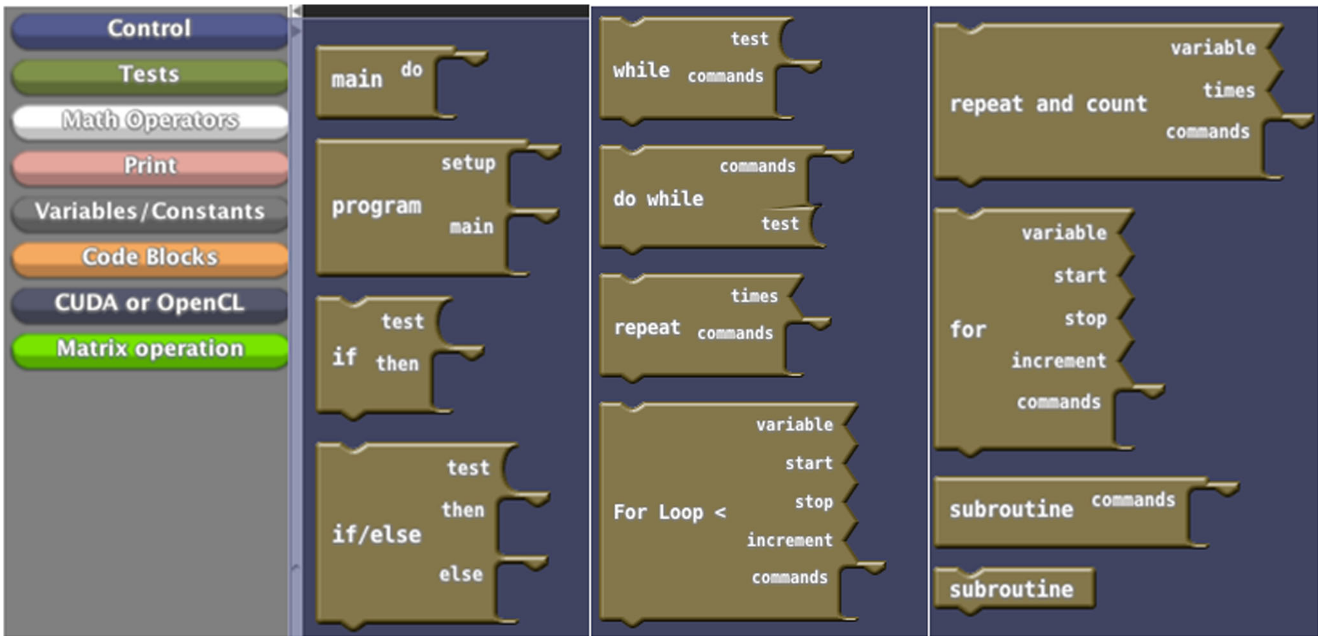
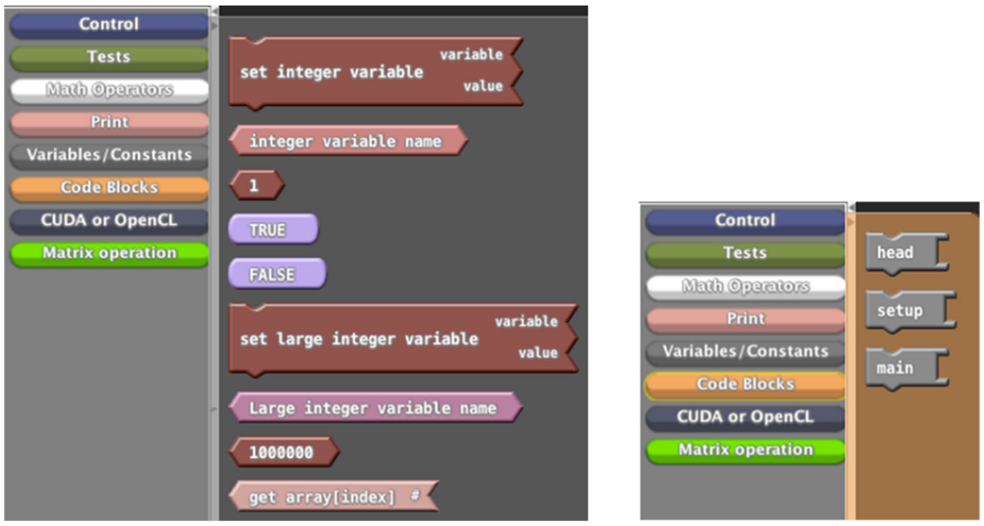


Figure 2 Control Blocks of ANSI C Blocks.

Figure 3 ANSI C Block.



(a) Variable/Constants Blocks (b) Code Blocks

to array or matrix computations that are specified by algorithms. GPUBlocks can then translate block-based code to CUDA or OpenCL programs.

- A prototype visual programming tool *CUDABlock* that converts the OpenBlocks-based diagrams into CUDA programs has been implemented upon OpenBlocks and ArduBlock [8].
- This paper extends CUDABlock by incorporating OpenCL blocks in the frontend and an OpenCL code generator in the backend. Consequently, programmers can conveniently drag and connect GPUBlocks blocks and then generate OpenCL or CUDA programs.

The rest of the paper is organized as follows. Section 2 surveys the related work. Section 3 highlights the key components of the GPUBlocks tool. Section 4 presents the experimental results and Section 5 concludes this paper.

2 Related Work

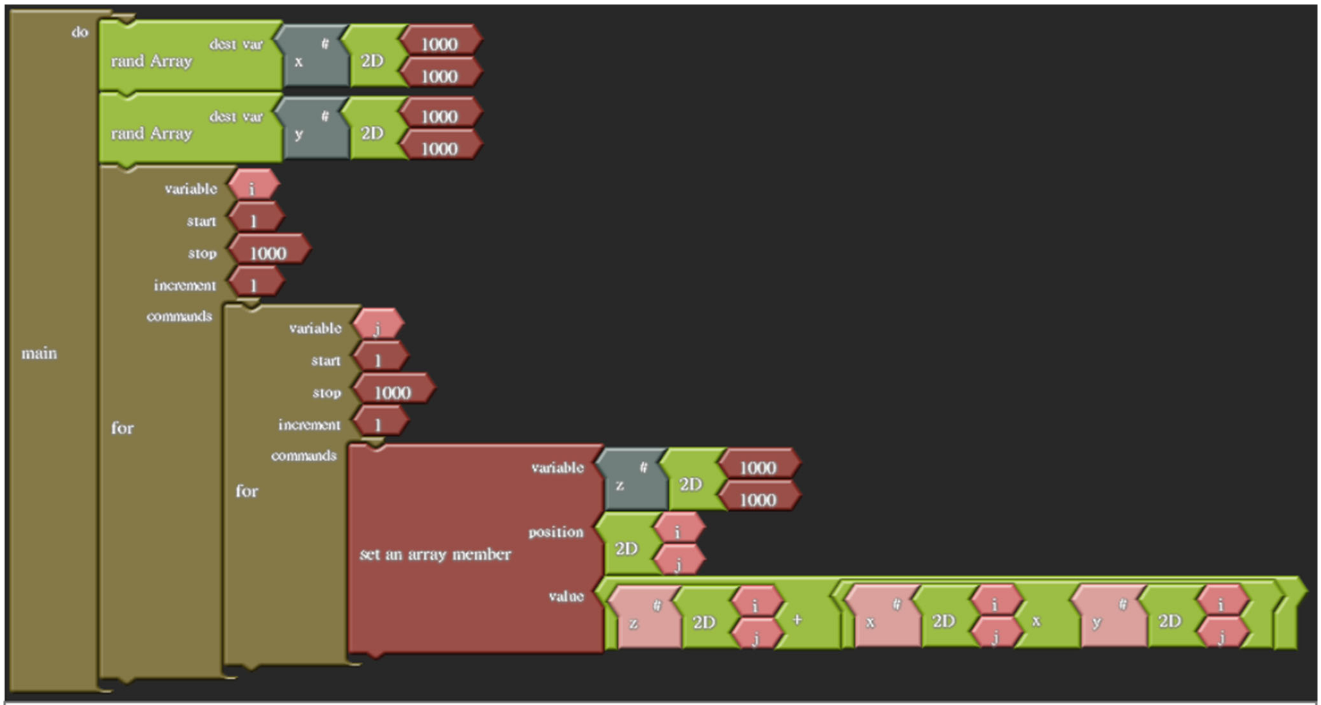
OpenCL is an open standard maintained by Khronos Group for the systems which incorporate with different types of computing devices, such as CPUs, GPUs, DSPs, and FPGAs [5]. With its platform-independent APIs, all OpenCL code is portable cross different devices. The standard is now supported by many hardware vendors, and with the device-specific drivers, the OpenCL-based data-parallel programs are able to take advantage of the computing power of underlying parallel hardware.

CUDA is a parallel computing platform and programming model invented by NVIDIA [13]. CUDA is a closed, data-parallel library, which is developed specifically to use a CUDA-enabled GPU for general purpose processing. Compared with OpenCL, CUDA is considered to be more efficient than the OpenCL counterpart on the NVIDIA platforms.

Overall, the above programming languages facilitate heterogeneous computing by abstracting different hardware architectures and generating the performance-oriented parallel code. Still, they are way too sophisticated for average programmers, who write programs for fun or focus on developing application algorithms.

In recent years, there are many attempts that have been made to develop visual programming frameworks for ease of the programming efforts and/or for computer education purpose, such as OpenBlocks [7], ArduBlock [1], NetLogo [12], StarLogo The Next Generation (TNG) [11], Scratch [10], Blockly [3], and Scratch Blocks [16]. Furthermore, several tools have been designed to support specific programming languages, such as Android App Inventor for Android system [9], and Hopscotch for Apple iOS [4].

Scratch is a graphical programming tool developed by MIT [10]. The educational purpose software aims to help young people learn to think creatively. It can be used to build interactive stories, games, and animations. Blockly [3], which is a Google's project, is an open source library that creates the virtual code editor in the form of web pages and Android apps. Also, Blockly helps create the programs with its visual code editor, and users do not worry about the language syntax. It supports the generation of various codes, including JavaScript, Python, PHP, Lua, and Dart, where each language has its own code generator, converting the diagrams into the corresponding program codes. Based on Blockly, Scratch Blocks [16] is a new development project for building creative learning tools for young people. The ongoing

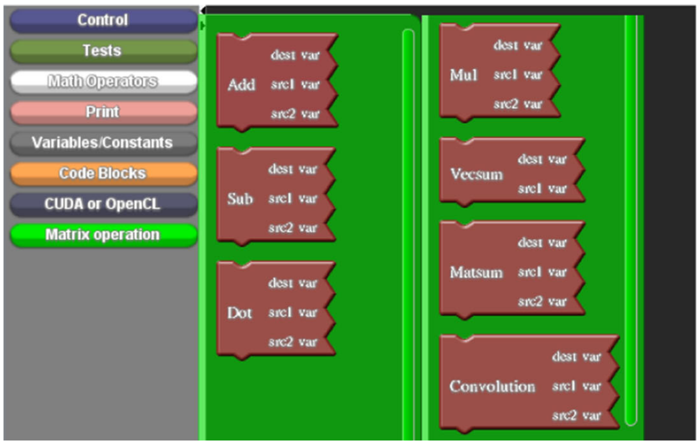


```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>

void matgen(float* a, int n)
{
  int i;
  for(i = 0; i < n; i++) {
    a[i] = (float) rand() / RAND_MAX +
      (float) rand() / (RAND_MAX * RAND_MAX);
  }
}

float *x= (float*)malloc(sizeof(float)*1000*1000);
float *y= (float*)malloc(sizeof(float)*1000*1000);
int i = 0 ;
int j = 0 ;
float *z= (float*)malloc(sizeof(float)*1000*1000);

void setup()
{
  for (int i=0;i<1000*1000;i++) x[i]=0;
  for (int i=0;i<1000*1000;i++) y[i]=0;
  for (int i=0;i<1000*1000;i++) z[i]=0;
}

int main()
{
  setup();
  matgen(x,1000*1000);
  matgen(y,1000*1000);
  for (i= (1) ; i <= (1000) ; i+= (1) ) {
    for (j= (1) ; j <= (1000) ; j+= (1) ){
      z[1000*(i - 1)+j - 1] = ( z[ 1000 * (i - 1) + (j - 1)] + ( x[ 1000 * (i -
1) + (j - 1)] * y[ 1000 * (i - 1) + (j - 1)] ) ) ;
    }
  }
  // matfree(float* a), which frees the allocated memory, pointed by a.
  matfree(x); matfree(y); matfree(z);
  return 0;
}
```

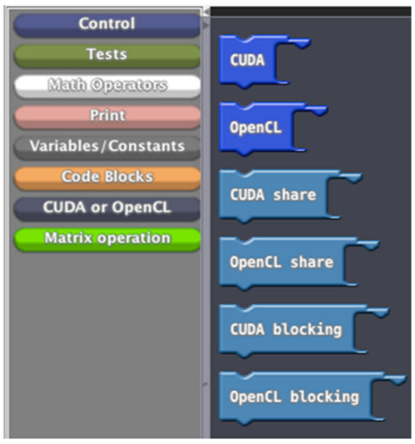
Figure 4 Example of ANSI C Blocks (top) and Its Corresponding C Code (bottom).

project aims to develop the next generation of graphical programming blocks, based on the collaboration between Google and the Scratch team from MIT. Currently, the developer preview code is available on the project website.

CUDABlock is a visual programming tool which converts the OpenBlocks-based diagrams into CUDA programs [8]. CUDABlock has a similar programming interface to ArduBlock [1], which is a graphic programming language for Arduino [2]. This work extends CUDABlock by incorporating OpenCL blocks in the frontend and an OpenCL code generator in the backend. Consequently, programmers can conveniently drag and connect GPUBlocks blocks and then generate OpenCL or CUDA programs.


Control
Tests
Math Operators
Print
Variables/Constants
Code Blocks
CUDA or OpenCL
Matrix operation
Add dest var src1 var src2 var
Sub dest var src1 var src2 var
Dot dest var src1 var src2 var
Mul dest var src1 var src2 var
Vecsum dest var src1 var
Matsum dest var src1 var src2 var
Convolution dest var src1 var src2 var

(a) Array and Matrix Computations



(b) CUDA and OpenCL Blocks

Figure 5 CUDA and OpenCL Blocks.

3 GPUBlocks

GPUBlocks has been developed based on the OpenBlocks framework by adding sets of new blocks for CUDA and OpenCL programming: *ANSI C Blocks*, *CUDA Blocks*, and *OpenCL Blocks*.

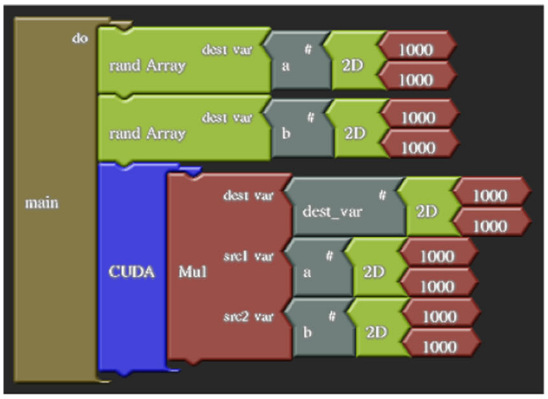
3.1 OpenBlocks Framework Overview

OpenBlocks is a Java-based visual programming tool that helps facilitate programming tasks by stacking the pre-defined *blocks* [7]. Programmers simply drag *code blocks* to denote specific actions, and then connect the selected blocks according to algorithms. Figure 1 shows an example of the OpenBlocks programming environment. The visual programming environment is divided into three areas. The top-left area allows users to choose from different blocks for programming, whereas the bottom-left region displays the different categories of the available blocks. The displaying area on the right is the main place for visual programming by dragging the blocks on the left and connecting them on the right.

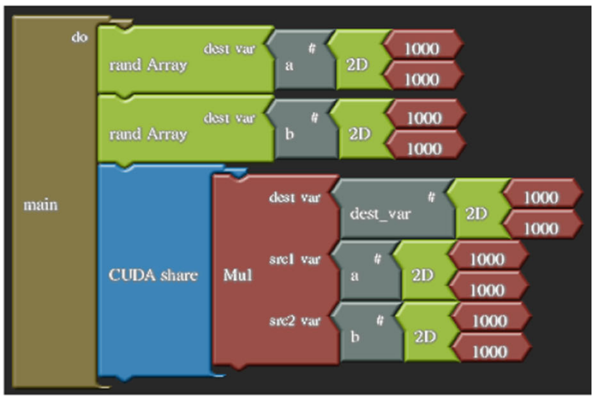
3.2 ANSI C Blocks

Five sets of ANSI C blocks have been integrated into GPUBlocks for C programming, namely Control, Test, Math, System IO, and Variables/Constants:

- *Control* blocks denote the control-flow related constructs in C, as shown in Fig. 2.
- *Test* blocks evaluate boolean expressions.
- *Math* blocks represent pre-built mathematical functions.
- *System IO* blocks refer to services offered by the host system, currently only the *print* function is implemented.


do
main
rand Array dest var # a 2D 1000 1000
rand Array dest var # b 2D 1000 1000
CUDA
Mul dest var dest_var # 2D 1000 1000
src1 var # a 2D 1000 1000
src2 var # b 2D 1000 1000

(a) Basic CUDA Program.


do
main
rand Array dest var # a 2D 1000 1000
rand Array dest var # b 2D 1000 1000
CUDA share
Mul dest var dest_var # 2D 1000 1000
src1 var # a 2D 1000 1000
src2 var # b 2D 1000 1000

(b) Optimized CUDA Program with Tiling.

Figure 6 CUDA Examples.

```
__global__ static void matMultCUDA_nonblock(const float* a, size_t lda,
const float* b, size_t ldb,
float* c, size_t ldc, int n)
{
   const int tid = threadIdx.x;
   const int bid = blockIdx.x;
   const int idx = bid * blockDim.x + tid;
   const int row = idx / n;
   const int column = idx % n;
   int i;
   if(row < n && column < n) {
      float t = 0;
      for(i = 0; i < n; i++) {
         t += a[row * lda + i] * b[i * ldb + column];
      }
      c[row * ldc + column] = t;
   }
}
```

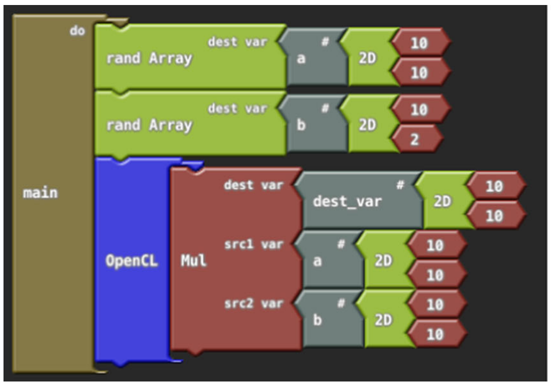
```
#define NUM_THREADS 256
clock_t matmultCUDA_nonblock(const float* a, int lda, const float* b, int ldb,
float* c, int ldc, int n)
{
   float *ac, *bc, *cc;
   clock_t start, end;
   start = clock();
   cudaMalloc((void**) &ac, sizeof(float) * n * n);
   cudaMalloc((void**) &bc, sizeof(float) * n * n);
   cudaMalloc((void**) &cc, sizeof(float) * n * n);
   cudaMemcpy2D(ac, sizeof(float) * n, a, sizeof(float) * lda,sizeof(float) * n, n,
cudaMemcpyHostToDevice);
   cudaMemcpy2D(bc, sizeof(float) * n, b, sizeof(float) * ldb, sizeof(float) * n,
n, cudaMemcpyHostToDevice);
   int blocks = (n + NUM_THREADS - 1) / NUM_THREADS;
   matMultCUDA_nonblock<<<blocks*n, NUM_THREADS>>>(ac, n, bc, n,
cc, n, n);
   cudaMemcpy2D(c, sizeof(float) * ldc, cc, sizeof(float) * n, sizeof(float) * n,
n, cudaMemcpyDeviceToHost);
   cudaFree(ac);
   cudaFree(bc);
   cudaFree(cc);
   end = clock();
   return end - start;
}
```

Figure 7 Generated CUDA Code of Fig. 6a: CUDA Kernel (left) and Host Code (right).

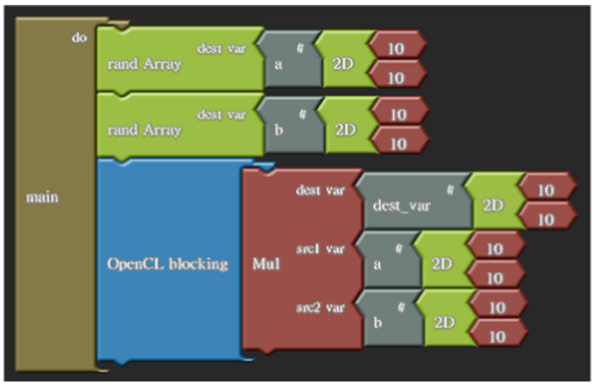
```
#define BLOCK_SIZE 32
__global__ static void matMultCUDA_blocking(const float* a, size_t lda, const float* b,
size_t ldb, float* c, size_t ldc, int n)
{
   __shared__ float matA[BLOCK_SIZE][BLOCK_SIZE];
   __shared__ float matB[BLOCK_SIZE][BLOCK_SIZE];
   const int tidc = threadIdx.x;
   const int tidr = threadIdx.y;
   const int bidc = blockIdx.x * BLOCK_SIZE;
   const int bidr = blockIdx.y * BLOCK_SIZE;
   int i, j;
   float results = 0;
   float comp = 0;
   for(j = 0; j < n; j += BLOCK_SIZE) {
      matA[tidr][tidc] = a[(tidr + bidr) * lda + tidc + j];
      matB[tidr][tidc] = b[(tidr + j) * ldb + tidc + bidc];
      __syncthreads();
      for(i = 0; i < BLOCK_SIZE; i++) {
         float t;
         comp -= matA[tidr][i] * matB[i][tidc];
         t = results - comp;
         comp = (t - results) + comp;
         results = t;
      }
      __syncthreads();
   }

   c[(tidr + bidr) * ldc + tidc + bidc] = results;
}
```

```
clock_t matmultCUDA_blocking(const float* a, int lda, const float* b, int ldb, float* c, int
ldc, int n)
{
   float *ac, *bc, *cc;
   clock_t start, end;
   size_t pitch_a, pitch_b, pitch_c;
   int newn = ((n + BLOCK_SIZE - 1) / BLOCK_SIZE) * BLOCK_SIZE;
   start = clock();
   cudaMallocPitch((void**) &ac, &pitch_a, sizeof(float) * newn, newn);
   cudaMallocPitch((void**) &bc, &pitch_b, sizeof(float) * newn, newn);
   cudaMallocPitch((void**) &cc, &pitch_c, sizeof(float) * newn, newn);
   cudaMemset(ac, 0, pitch_a * newn);
   cudaMemset(bc, 0, pitch_b * newn);
   cudaMemcpy2D(ac, pitch_a, a, sizeof(float) * lda, sizeof(float) * n, n,
cudaMemcpyHostToDevice);
   cudaMemcpy2D(bc, pitch_b, b, sizeof(float) * ldb, sizeof(float) * n, n,
cudaMemcpyHostToDevice);
   int bx = (n + BLOCK_SIZE - 1) / BLOCK_SIZE;
   dim3 blocks(bx, bx);
   dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
   matMultCUDA_blocking<<<blocks, threads>>>(ac, pitch_a / sizeof(float), bc, pitch_b /
sizeof(float), cc, pitch_c / sizeof(float), n);
   cudaMemcpy2D(c, sizeof(float) * ldc, cc, pitch_c, sizeof(float) * n, n,
cudaMemcpyDeviceToHost);
   cudaFree(ac);
   cudaFree(bc);
   cudaFree(cc);
   end = clock();
   return end - start;
}
```

Figure 8 Optimized CUDA Code of Fig. 6b: CUDA Kernel (left) and Host Code (right).



(a) Basic OpenCL Program.



(b) Optimized OpenCL Program with Tiling.

Figure 9 Examples of OpenCL Blocks Programs.

- *Variable/Constants* blocks are used to declare constants and variables, shown in Fig. 3a.

In addition, there is another class of blocks, *Code Blocks*, which helps build customized C codes by allowing users to define specialized code blocks. As shown in Fig. 3b, there are three buttons, *head*, *setup*, and *main* in the Code Blocks. Users can make customized codes for these buttons. In particular, head button is for including a header file or function/variable declarations, setup button can be dragged into the main function for function/variable initialization, and main button allows the user to specify the customized statements.

Figure 4 depicts an example of programming a 2D array computation. The top part of the figure depicts the blocks for the initialization and computations of a 2D array, while the bottom part shows its corresponding C statements that are generated by GPUBlocks. This example clearly shows that the above operations can be done easily with two nested for-loop blocks and a variable block.

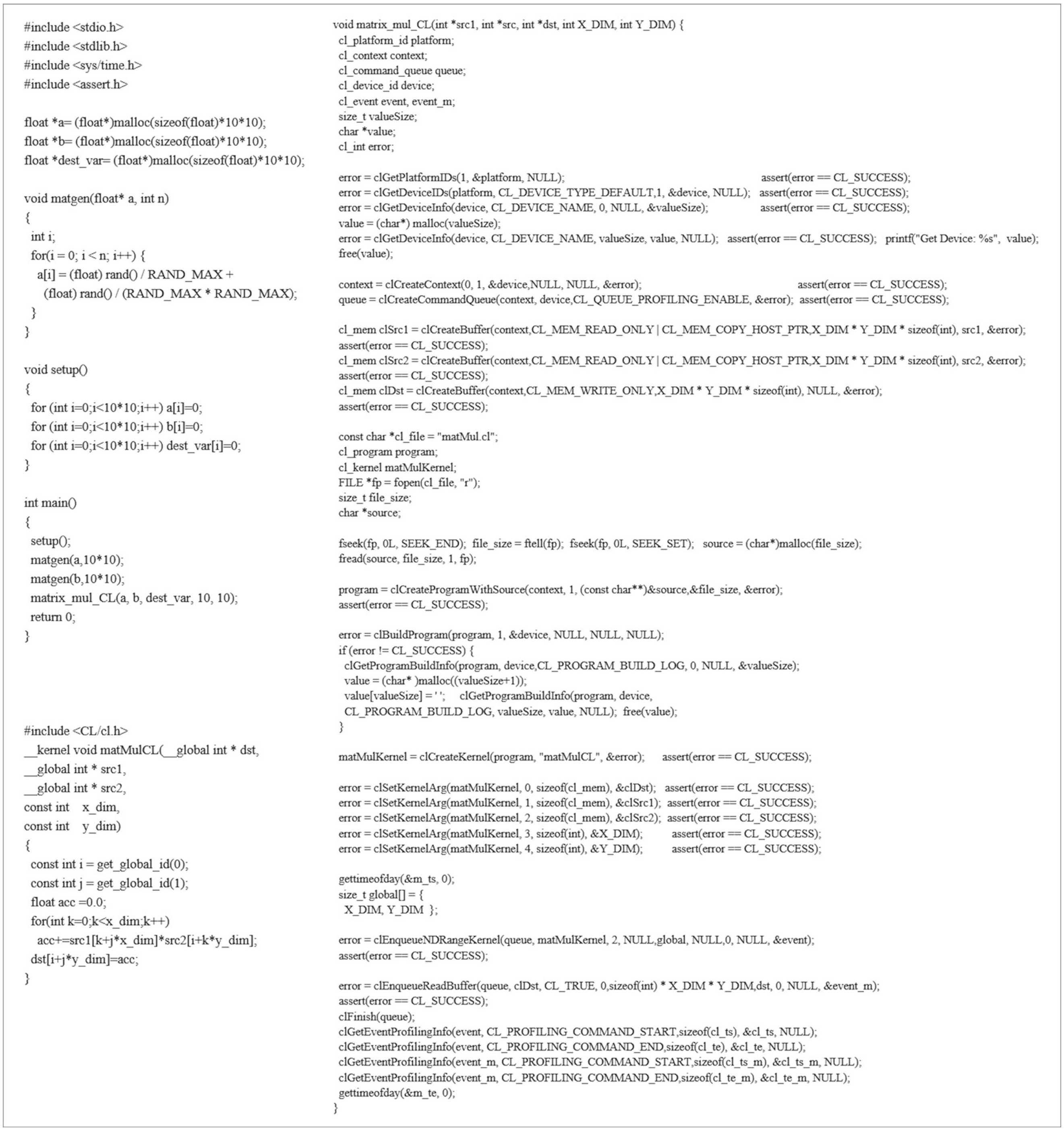
3.3 CUDA and OpenCL Blocks

GPUBlocks supports basic array and matrix operations of linear algebra, e.g., addition, subtraction, multiplication, and convolution operations, as listed in Fig. 5a. The default setting is that these operations will be translated into a sequential C program. When a CUDA block or an OpenCL block is placed before these array and matrix operations, GPUBlocks will convert the block program into a CUDA program or an OpenCL program, respectively. In other words, it is straightforward to switch among C, CUDA, and OpenCL by dragging an appropriate language block and placing it before array and matrix operations.

In addition to CUDA and OpenCL blocks, Fig. 5b illustrates that GPUBlocks also includes blocks of two basic and commonly used optimization techniques in both CUDA and OpenCL, i.e. *share* blocks for shared memory caching and a *blocking* blocks for data tiling. When a *share* block is used, the backend of GPUBlocks will generate a CUDA or OpenCL code that allocates data in the GPU shared memory. As shared memory is on-chip, it is much faster than local and global memory, roughly 100x lower than uncached global memory latency. Therefore, speedups generally can be observed when this optimization is applied. Furthermore, additional speedups might be achieved if a *blocking* block is chosen, since tiling (or block) is a commonly used programming pattern that partitions data in order to operate in well-sized blocks whose size is small enough to be staged in shared memory.

CUDA Example This section uses a matrix multiplication as an example to illustrate GPUBlocks programs and their corresponding CUDA code. Two GPUBlocks variable blocks are first constructed to initialize two random matrixes a and b, and then a matrix multiplication block and a CUDA block are added to specify performing the matrix multiplication computation $c = a \times b$ in CUDA, as illustrated in Fig. 6a. GPUBlocks then converts the code into its corresponding kernel code and host program, as listed in Fig. 7. This example has demonstrated that GPUBlocks is an intuitive approach that can significantly reduce the complexity of parallel programming in CUDA.

Performing optimizations on CUDA programs is easy and straightforward in GPUBlocks by simply replacing the plain CUDA blocks with the specific optimization blocks. Figure 6b shows that only one block needs to be replaced in order to utilize shared memory to optimize the CUDA code, and Fig. 8 lists the optimized CUDA code that is generated by GPUBlocks. This example has shown that this intuitive approach can effectively reduce the complexity of writing optimized CUDA programs.



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>

float *a= (float*)malloc(sizeof(float)*10*10);
float *b= (float*)malloc(sizeof(float)*10*10);
float *dest_var= (float*)malloc(sizeof(float)*10*10);

void matgen(float* a, int n)
{
 int i;
 for(i = 0; i < n; i++) {
   a[i] = (float) rand() / RAND_MAX +
     (float) rand() / (RAND_MAX * RAND_MAX);
 }
}

void setup()
{
 for (int i=0;i<10*10;i++) a[i]=0;
 for (int i=0;i<10*10;i++) b[i]=0;
 for (int i=0;i<10*10;i++) dest_var[i]=0;
}

int main()
{
 setup();
 matgen(a,10*10);
 matgen(b,10*10);
 matrix_mul_CL(a, b, dest_var, 10, 10);
 return 0;
}
```

```
#include <CL/cl.h>
__kernel void matMulCL(__global int * dst,
__global int * src1,
__global int * src2,
const int    x_dim,
const int    y_dim)
{
 const int i = get_global_id(0);
 const int j = get_global_id(1);
 float acc =0.0;
 for(int k=0;k<x_dim;k++)
   acc+=src1[k+j*x_dim]*src2[i+k*y_dim];
 dst[i+j*y_dim]=acc;
}
```

```
void matrix_mul_CL(int *src1, int *src, int *dst, int X_DIM, int Y_DIM) {
 cl_platform_id platform;
 cl_context context;
 cl_command_queue queue;
 cl_device_id device;
 cl_event event, event_m;
 size_t valueSize;
 char *value;
 cl_int error;

 error = clGetPlatformIDs(1, &platform, NULL);                                    assert(error == CL_SUCCESS);
 error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_DEFAULT,1, &device, NULL);  assert(error == CL_SUCCESS);
 error = clGetDeviceInfo(device, CL_DEVICE_NAME, 0, NULL, &valueSize);           assert(error == CL_SUCCESS);
 value = (char*) malloc(valueSize);
 error = clGetDeviceInfo(device, CL_DEVICE_NAME, valueSize, value, NULL);   assert(error == CL_SUCCESS);   printf("Get Device: %s",  value);
 free(value);

 context = clCreateContext(0, 1, &device,NULL, NULL, &error);                                 assert(error == CL_SUCCESS);
 queue = clCreateCommandQueue(context, device,CL_QUEUE_PROFILING_ENABLE, &error);  assert(error == CL_SUCCESS);

 cl_mem clSrc1 = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,X_DIM * Y_DIM * sizeof(int), src1, &error);
 assert(error == CL_SUCCESS);
 cl_mem clSrc2 = clCreateBuffer(context,CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,X_DIM * Y_DIM * sizeof(int), src2, &error);
 assert(error == CL_SUCCESS);
 cl_mem clDst = clCreateBuffer(context,CL_MEM_WRITE_ONLY,X_DIM * Y_DIM * sizeof(int), NULL, &error);
 assert(error == CL_SUCCESS);

 const char *cl_file = "matMul.cl";
 cl_program program;
 cl_kernel matMulKernel;
 FILE *fp = fopen(cl_file, "r");
 size_t file_size;
 char *source;

 fseek(fp, 0L, SEEK_END);  file_size = ftell(fp);  fseek(fp, 0L, SEEK_SET);   source = (char*)malloc(file_size);
 fread(source, file_size, 1, fp);

 program = clCreateProgramWithSource(context, 1, (const char**)&source,&file_size, &error);
 assert(error == CL_SUCCESS);

 error = clBuildProgram(program, 1, &device, NULL, NULL, NULL);
 if (error != CL_SUCCESS) {
  clGetProgramBuildInfo(program, device,CL_PROGRAM_BUILD_LOG, 0, NULL, &valueSize);
  value = (char* )malloc((valueSize+1));
  value[valueSize] = ' ';     clGetProgramBuildInfo(program, device,
  CL_PROGRAM_BUILD_LOG, valueSize, value, NULL);  free(value);
 }

 matMulKernel = clCreateKernel(program, "matMulCL", &error);      assert(error == CL_SUCCESS);

 error = clSetKernelArg(matMulKernel, 0, sizeof(cl_mem), &clDst);   assert(error == CL_SUCCESS);
 error = clSetKernelArg(matMulKernel, 1, sizeof(cl_mem), &clSrc1);  assert(error == CL_SUCCESS);
 error = clSetKernelArg(matMulKernel, 2, sizeof(cl_mem), &clSrc2);  assert(error == CL_SUCCESS);
 error = clSetKernelArg(matMulKernel, 3, sizeof(int), &X_DIM);        assert(error == CL_SUCCESS);
 error = clSetKernelArg(matMulKernel, 4, sizeof(int), &Y_DIM);        assert(error == CL_SUCCESS);

 gettimeofday(&m_ts, 0);
 size_t global[] = {
  X_DIM, Y_DIM  };

 error = clEnqueueNDRangeKernel(queue, matMulKernel, 2, NULL,global, NULL,0, NULL, &event);
 assert(error == CL_SUCCESS);

 error = clEnqueueReadBuffer(queue, clDst, CL_TRUE, 0,sizeof(int) * X_DIM * Y_DIM,dst, 0, NULL, &event_m);
 assert(error == CL_SUCCESS);
 clFinish(queue);
 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,sizeof(cl_ts), &cl_ts, NULL);
 clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,sizeof(cl_te), &cl_te, NULL);
 clGetEventProfilingInfo(event_m, CL_PROFILING_COMMAND_START,sizeof(cl_ts_m), &cl_ts_m, NULL);
 clGetEventProfilingInfo(event_m, CL_PROFILING_COMMAND_END,sizeof(cl_te_m), &cl_te_m, NULL);
 gettimeofday(&m_te, 0);
}
```

Figure 10 Generated OpenCL Code of Fig. 9a, Kernel (bottom-left) and Host Code (right).

OpenCL Example Figure 9 depicts the same matrix multiplication example in OpenCL, which looks almost identical to the CUDA example shown Fig. 6. The unoptimized code in Fig. 9a is translated into the OpenCL code presented in Fig. 10. While generating OpenCL programs, GPUBlocks makes the following assumptions unless otherwise specified. First, GPUBlocks uses the first computing device that could be found by the OpenCL runtime to perform the computation of the generated code. Second, GPUBlocks inserts the *assertion functions* which OpenCL function would call to ensure that the parallel code executes as expected. By default, the debugging feature is enabled in order to notify programmers if there are problems during the execution of the generated programs. Third, GPUBlocks also injects the performance debugging code for the purpose of performance analysis. However, the default settings

```
__kernel void matrixMul(__global float* C,
                        __global float* A,
                        __global float* B, int wA, int wB)
{
    int bx = get_group_id(0);
    int by = get_group_id(1);
    int tx = get_local_id(0);
    int ty = get_local_id(1);
    float Csub = 0.0;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd    = aBegin + wA - 1;
    int aStep   = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep   = BLOCK_SIZE * wB;
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep)
    {
        __local float As[BLOCK_SIZE][BLOCK_SIZE];
        __local float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

Figure 11 Generated OpenCL kernel code illustrated in Fig. 9b.

above can be changed via the configuration file, and the new configuration will take effect when GPUBlocks is re-started.

Figure 11 depicts the data tiling code of the matrix multiplication program in Fig. 9b. As shown in the OpenCL kernel code, the current thread id is obtained first to calculate the boundary of the data block to be processed. Inside the for loop, local memory buffers are used to keep the data block in the device local memory, and data required for the matrix multiplication are read from the local buffers, which accelerates the program performance. While data tiling is a common technique in parallel computing, it is tedious work, and costs significant amount of time for average programmers. GPUBlocks simplifies the optimization process, and helps generate the optimization code on-the-fly.

4 Experiment Results

Several array and matrix operations have been performed to evaluate the efficiency of the generated OpenCL and CUDA programs. Microbenchmarks, as shown in Fig. 12, have been executed on the Linux/x86 system with the Intel Xeon E5506 Processor, where their corresponding CUDA and OpenCL programs have been tested on the NVIDIA Geforce GTX 1080 (CUDA SDK 8.0 and OpenCL 1.2). Each program is executed with the following four configurations.

- *cpu*. The baseline performance is the elapsed time of sequential C code on the main processor.
- *gpu*. The total time of the generated CUDA or OpenCL programs that have spent on both main processor and GPU.
- *shared*. The total time that have been spent on both main processor and GPU by the generated CUDA or OpenCL programs with shared memory caching.
- *blocking*. The total time that have been spent on both main processor and GPU by the generated CUDA or OpenCL programs with the tiling optimization.

Figures 13 and 14 depict the experimental results of the CUDA and OpenCL programs, respectively. Note that the sequential C code, denoted as *cpu*, is the baseline configuration as an indicator to show the performance differences between the CUDA and OpenCL

Figure 12 The microbenchmarks and their input data sizes.

Algorithm name	input
ADD	10240*10240
SUB	10240*10240
ScalarMUL	10240*10240
MUL	10240*10240
convolution	4096*4096 filter size:3*3
Vecsum	1024*20*1024*20
volve	1024*1024
matsum	1024*12 * 10240*12
mulsum	10240*10240

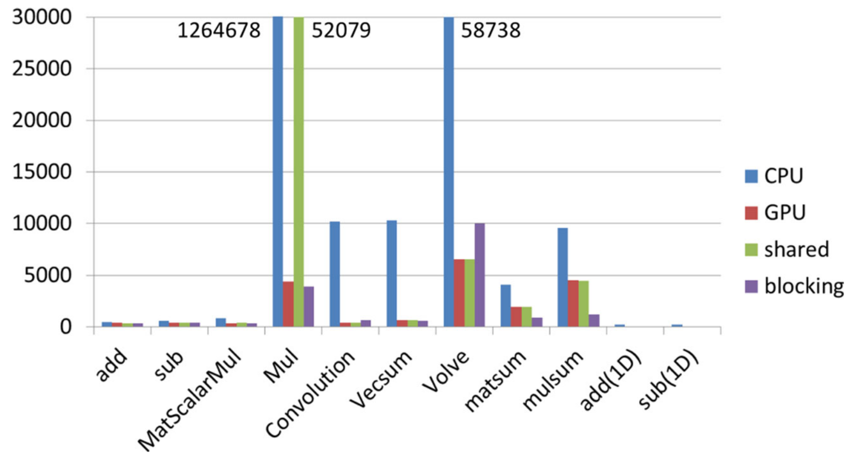


Figure 13 CUDA Performance on NVIDIA GPU.

versions. Overall, the program performance delivered by the NVIDIA device outperforms that by the AMD device in our experiments. In addition, the optimized CUDA/OpenCL versions are faster than those without optimizations. Nevertheless, optimizations may lead to poor performance, e.g., shared optimization for *matmul* and blocking for *Volve*.

The slowdown of the optimized codes could be attribute to the program behaviors, and software/hardware interactions. For example, while the shared optimization copying the data to the local buffers prior to the computation, the matrix multiplication does not reuse the localized data, and hence the data copying adds the overhead. The delivered performance of the blocking optimization would depend on the number of concurrent threads, which would vary across different hardware/software combinations, and intensive experiments should be done to explore the best configuration, e.g., the thread number.

In current stage, performance tuning for the best configuration of the generated program is not the focus of this work. Still, we develop some facilities to help profile the performance of the converted programs, and hopefully, programmers could use them to tweak the program performance.

5 Conclusion

This paper introduced a visual programming tool GPUBlocks for CUDA and OpenCL programming. This tool could help beginners and average programmers to implement CUDA and OpenCL programs by simply dragging and connecting blocks. The generated CUDA and OpenCL programs could be used directly to perform computations on GPU, or served as the first draft of CUDA and OpenCL kernels which would be further optimized manually.

Acknowledgment This work was supported in part by the Ministry of Science and Technology of Taiwan under Grants MOST 104-2622-8-002-002, MOST 105-2622-8-002-002, MOST 104-2221-E-011-007, and MOST 105-2221-E-011-069-MY3, and sponsored by MediaTek Inc., Hsin-chu, Taiwan.

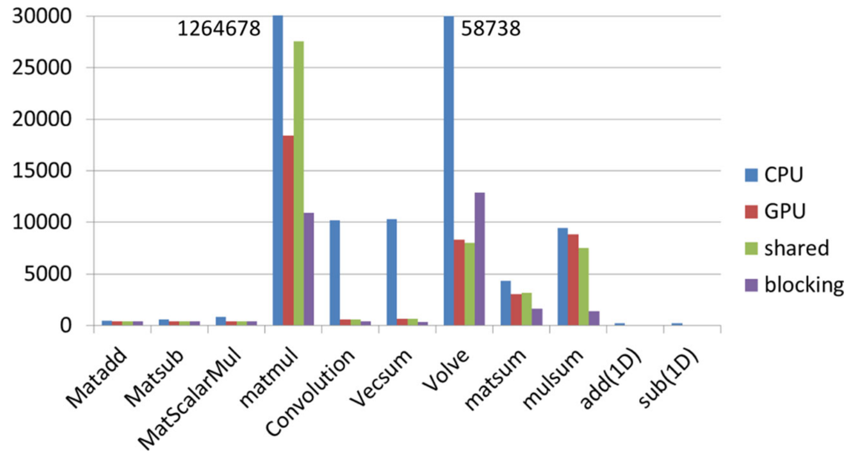


Figure 14 OpenCL Performance on AMD GPU.

References

1. ArduBlock. ArduBlock: A Graphical Programming Language for Arduino. http://blog.ardublock.com.
2. Banzi, M. (2008). Getting started with arduino. Make Books - Imprint of: O'Reilly Media.
3. Google. Blockly. https://developers.google.com/blockly/.
4. Hopscotch. Hopscotch: A programming tool for developing apps for ios devices https://www.gethopscotch.com/.
5. Khronos Group. OpenCL: The open standard for parallel programming of heterogeneous systems https://www.khronos.org/opencl/.
6. Khronos Group. The OpenMP API specification for parallel programming http://openmp.org/wp/.
7. Klopfer, E. (2007). *OpenBlocks: An extendable framework for graphical block programming systems*. Cambridge: PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
8. Lin, H.-H., Tu, C.-H., Hwang, Y.-S. (2015). CUDABlock: A GUI programming tool for CUDA. In *2015 International Workshop on Embedded Multicore Systems, 44th International Conference on Parallel Processing Workshops, ICPPW, 2015* (pp. 37–42).
9. MIT. MIT App Inventor: A blocks-based programming tool for developing apps for android devices http://appinventor.mit.edu/explore/.
10. MIT. Scratch: A free desktop and online multimedia authoring tool https://scratch.mit.edu/.
11. MIT. Starlogo TNG: The next generation of starlogo modeling and simulation software http://education.mit.edu/projects/starlogo-tng.
12. NetLogo. Netlogo: a multi-agent programmable modeling environment http://ccl.northwestern.edu/netlogo/.
13. NVIDIA. CUDA: A parallel computing platform and programming model http://www.nvidia.com/object/cuda_home_new.html.
14. OpenMPI. Open MPI: Open source high performance computing http://www.open-mpi.org/.
15. OpenVX. OpenVX: Protable, power-efficient vision processing https://www.khronos.org/openvx/.
16. Scratch for Developers. Scratch Blocks https://scratch.mit.edu/developers.

Yuan-Shin Hwang received the B.S. and M.S. degrees in electrical engineering from the National Tsing Hua University, Hsinchu, Taiwan in 1987 and 1989, respectively, and the M.S. and Ph.D. degrees in computer science in 1994 and 1998 from the University of Maryland at College Park. He is a professor in the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan. His research interests include parallel and distributed computing, parallel architectures, parallelizing compilers, and programming languages.

Hsih-Hsin Lin received the B.S. and M.S. degrees in computer science from National Taiwan University of Science and Technology, Taiwan in 2014. He is an engineer in the Wistron Corporation, Taiwan. His research interests include program analysis, parallel programming, and compiler optimization.

Shen-Hung Pai received the B.S. degree in computer science from National Dong Hwa University in 2014, and the M.S. degree in computer science form National Taiwan University of Science and Technology, Taiwan in 2016. His research interests include parallel programming, and compiler optimization.

Chia-Heng Tu is an assistant professor with Department of Computer Science and Information Engineering (CSIE), National Cheng Kung University (NCKU). Before joining NCKU-CSIE, he worked as Postdoctoral Researcher in MEDIATEK-NTU Advanced Research Center, National Taiwan University (NTU) in 2015, and as R&D Manager in Institute for Information Industry from 2012 to 2015, after he completed his Ph.D. training from NTU in 2012. His research interests are developing tools (e.g., computer architecture simulators, performance analyzers/optimizers, and parallelizing compilers) for designing/optimizing specialized computer systems.