A Case Study for GPGPU Program Verification

Steven de Heus
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
stevendeheus@gmail.com

ABSTRACT

The goal of this research is to design a concurrent program that computes the edit distance between two strings and then verify the correctness of the program. The program is implemented in OpenCL and specification and verification is done using *permission-based separation logic*. The logic used for the (manual) verification is the same as used in the VerCors tool set, which is currently under development. The results of this research will assist in further development of VerCors.

Keywords

OpenCL, program verification, edit distance, case study, GPGPU program, permission-based separation logic, VerCors tool set

1. INTRODUCTION

Graphics Processing Units (GPUs) are used increasingly for tasks other than processing graphics. Because of the typical architecture of GPUs (hundreds to thousands of cores), they lend themselves well for concurrent programs. GPGPU (General-Purpose GPU) programming is interesting because programs that can divide workload over many threads can run much more efficiently on GPUs than on CPUs. Since GPU programming is on the rise, it is important to be able to reason about the correctness of such programs.

With the expansion of GPGPU programming and verification comes a demand for more verification case studies. For example, a tool set called VerCors [2] is being developed. This tool set reasons about multithreaded programs using permission-based separation logic. For the development of this tool set, more GPGPU program verification case studies that use the same type of specification and verification logic are needed [6]. This research will provide such a case study.

A common type of algorithm that can be implemented on GPUs is dynamic programming. With dynamic programming, a problem is broken into multiple similar subproblems. GPGPU programming is well-suited for dynamic programming because in GPGPU programming the same set of instructions is sent to multiple cores of the GPU. This concept is better known as Single Program Multiple Data (SPMD). A good example of a problem that can be solved with dynamic programming is the edit distance problem. The edit distance is the minimal cost to convert a word into another word using the weighted actions insertion, deletion and transformation of characters. A solution to the edit distance problem can be modified to also solve other problems. An example is Levenshtein [8], a variation of edit distance where all costs are always one. Other problems that can be solved with dynamic programming with similar data dependencies between the subproblems can also be solved with similar algorithms. Since edit distance provides a good example of dynamic programming and the solution can be applied to other problems as well, it will be used for this case study of GPGPU program verification.

A GPGPU program that finds the edit distance between two strings will be implemented and manually verified using permission-based separation logic. This research will set an example for future work on GPGPU program specification and verification and more specifically, the VerCors tool set.

2. RESEARCH QUESTIONS

The main research question is:

How can we reason about the correctness of GPGPU programs?

This research question is divided into multiple sub-questions:

1. What are the formal requirements the program needs to meet?
2. How can we implement the program on a GPU?
3. How can we prove that the program meets the requirements?
4. How can we optimize the performance of the program?

3. BACKGROUND

3.1 Edit distance

The edit distance between two strings A and B is the minimal amount of work required to transform A into B. For the transformation, characters may be inserted, deleted or transformed into other characters. These three actions have variable costs C_i (insertion), C_d (deletion) and C_t (transformation). Examples of applications of edit distance are spell checkers, autofill for search, correction systems for optical character recognition and DNA comparison [11].

Below is an example of input for the program:

$$\begin{aligned} A &= \text{‘abcde’} \\ B &= \text{‘abde’} \\ C_i &= 5 \\ C_d &= 10 \\ C_t &= 15 \end{aligned}$$

The output the program should return for this input is 5, because A can be transformed into B by inserting the character ‘c’ and the cost of inserting a character is 5.

3.2 OpenCL

OpenCL [7] is a programming language that can be used to write concurrent programs that can run on both GPUs and CPUs. It is compatible with most hardware (most importantly, both Nvidia and AMD). OpenCL is a low level programming language based on C. Another programming language commonly used for GPGPU programming is CUDA [1]. CUDA is similar to OpenCL but runs on Nvidia devices exclusively. The programming language chosen for this research is OpenCL because it runs on more platforms.

3.3 GPGPU programming

For GPGPU programming it is important to consider the OpenCL memory model [7]. GPUs typically have hundreds of cores. The different cores or *work-items* are grouped together in *work-groups*. Work-items have access to *private memory*, *local memory* (shared by all work-items in the same work-group) and *global memory* (accessible by all work-groups). In order to optimize the performance of a program it is important to keep global memory usage to a minimum and instead work on the faster local/private memory as much as possible. All work-items used by the program receive the same set of instructions (kernel) but will execute them on different data depending on their global and local identifier (Single Program Multiple Data). Synchronization between work-items is achieved by using *barriers*. When a work-item reaches a barrier it waits until all other work-items in the same work-group reach the barrier before it continues. Synchronization between work-groups is hard to achieve and generally discouraged.

3.4 Permission-based separation logic

In standard separation logic [10], mutual exclusions mechanisms are used to ensure that only one thread at a time has write access to a shared memory location. However, this type of separation does not give enough freedom to reason about concurrent programs as it does not allow simultaneous reads to shared locations [6]. Permission-based separation logic uses numerical permission values ($0 < perm \leq 100$) to denote rights to a shared location [4]. The highest permission grants write access to a shared location ($perm(location, 100)$). Any value lower than 100 only grants read access. Since VerCors keeps track of which threads still hold read permissions, it is acceptable to use any value between 0 and 100 or just ‘p’ for read permissions ($perm(location, p)$). Permission-based separation logic guarantees lack of data-races: when a work-item writes to a location, no other work-items can read from or write to the same location.

4. RELATED WORK

Farivar et al. [5] have designed an efficient algorithm that computes edit distance on GPUs. The optimizations in this research, although written in CUDA, could be used for our own program. One of these optimizations is to split the problem into independent quadrants to reduce global memory usage.

Huisman and Mihelcic have done research on verification of GPU programs (using permission-based separation logic) [6]. Huisman et al. are also developing VerCors, a tool set that reasons about the correctness of concurrent programs.

Some automated GPU verification tools already exist. Examples are PUGpara [9] and GPUVerify [3]. These automated verifiers are based on verification methods different from permission-based separation logic. PUGpara uses *parameterized verification* and GPUVerify uses *synchronous, delayed visibility.*

5. IMPLEMENTATION

The overall idea is to divide the problem into smaller subproblems, each of them requiring the edit distance between a substring of A and a substring of B (both substrings start at the frist character of A and B respectively). The results of the subproblems can be stored in a $(n+1)*(m+1)$ matrix like in Figure 1. Each cell in the matrix corresponds with a combination of substrings. Let A[0..n] and B[0..m] be the two input strings with length n+1 and m+1, respectively, and let D(k, l) be the edit distance between the substrings A[0..k] and B[0..l] where $0 \leq k \leq n, 0 \leq l \leq m$. The costs for the actions *insert, delete* and *transform* are described as C_i, C_d, C_t respectively.

		s	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
s	1								
u	2								
n	3								
d	4								
a	5								
y	6								

Figure 1. First iteration of edit distance

The first row and column of the matrix can be filled easily. The first row is computed as D(0, l) $=C_i$ * l where 0 <= l <= n, because the minimal amount of work to transform the empty string into any other string is equal to the amount of characters in that string multiplied by the cost of inserting a character. The same goes for the first column, but with deletion instead of insertion: D(k, 0) = C_d * k where 0 <= k <= n. So if the costs for insertion and deletion are 1, the first row and column will both be [0, 1, 2, 3..] as shown in Figure 1.

When characters A[k] and B[l] are equal, the following statement holds: D(k, l) = D(k - 1, l - 1). The reason for this is that the transformation from A[0..k] to B[0..l] can be done with exactly the same set of actions as the transformation from A[0..k-1] to B[0..l-1], as the extra character at the end of both strings does not have any impact on the actions used. An example of this is D(1,1) as shown in figure 2. In Figure 2, the yellow cells indicate that there is now enough information available to compute the value for the corresponding subproblem.

When characters A[k] and B[l] are not equal, the value in the corresponding cell is the minimum of the three values: $D(k-1, l)+C_i, D(k, l-1)+C_d$ and $D(k-1, l-1)+C_t$. This way subproblems are solved by picking the most efficiently solved subproblems (with regard to the different costs of the actions). In figure 3 D(2,1) and D(1,2) are computed this way. This guarantees that after the final iteration the minimal amount of work needed to transform A into B is equal to the value D(n, m) (the lower right corner of the matrix). The behaviour of the algorithm can also be

		s	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
s	1	🟩 0	🟨						
u	2	🟨							
n	3								
d	4								
a	5								
y	6								

Figure 2. Second iteration of edit distance

expressed as a function:

$$D(k,l) = \begin{cases} D(k-1,l-1) & \text{if } A(k) = B(l) \\ min(D(k-1,l) + C_i, & \\ D(k,l-1) + C_d, & \\ D(k-1,l-1) + C_t) & \text{if } A(k) \neq B(l) \end{cases}$$

In figure 2 and 3 the data dependencies between the cells are shown in dark and light gray (green and yellow), meaning that in order to compute the cells in yellow, the values in the green cells must be known. These figures show that an entire diagonal at a time can be computed concurrently, but no more than that. To prevent desynchronization between work-items, a barrier is added after computing the value in each cell. Without these barriers it would be possible for different work-items to work in different diagonals at the same time, which could lead to data races when the same memory location is read by one work-item while another work-item is writing to that location.

		s	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
s	1	🟩 0	🟩 1	🟨					
u	2	🟩 1	🟨						
n	3	🟨							
d	4								
a	5								
y	6								

Figure 3. Third iteration of edit distance

In order to compute all values in the matrix, work-items are initialized as shown in figure 4. Each cell with bold borders indicates it is a starting position of one of the work-items, meaning one work-item will compute the value for that cell and then work its way through the matrix. In order to achieve this, the coordinates for the starting positions of the work-items are $((1 - mheight)/2 + wid + 1, (mheight - 1)/2 - wid + 1)$ where *mheight* is the height of the matrix and *wid* is the work-item id. In order to cover the entire matrix the work-items alternate between increasing the x and y coordinate after every iteration of the algorithm, resulting in the pattern shown in Figure 5. Figure 6 illustrates when the work-items come across barriers. A downside of this pattern is that not all work-items are always working on an existing cell, which means they are effectively idling. To illustrate this, empty rows were added in Figure 5 and 6

5.1 Kernel design

The kernel is the code that is run by all the work-items on the GPU. The kernel consists of three parts: initialization, the main loop and a wrap-up. During initialization each work-item retrieves its id and computes its starting coordinates. In each iteration of the main loop the work-items compute one diagonal in the matrix and increase their coordinates accordingly. During wrap-up a single work-item sends the result back to the host, where it will be printed to the console. Below the code is divided into snippets which are clarified individually. The full program can be found in the appendix.

This function serves as a workaround for the fact that OpenCL does not support two dimensional matrices. The programming is done mostly with two dimensional coordinates. When a value from the matrix needs to be accessed or stored, the two dimensional coordinates are translated to an array index using this function. This functions organizes the matrix in an array as a sequence of rows.

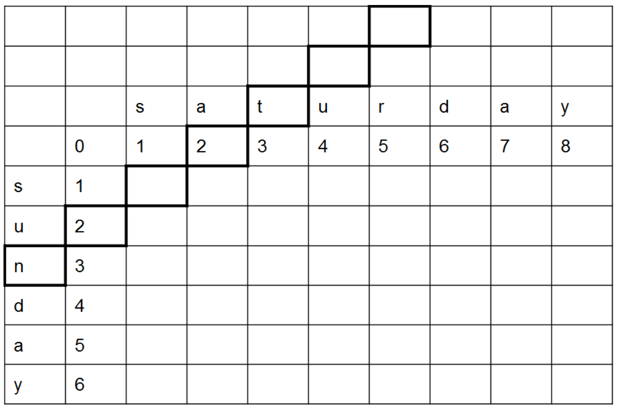
```
int to1D(int x, int y, int width) {
    return y*width + x;
}
```

Each work-item computes the x and y coordinates of their starting position using wid (work-group id). Only one work-item starts at a position where computation is necessary (at (1,1)), all the other work-items have their starting position in the first row or column or in a non-existent position, as shown in figure 4. The array *dimensions* holds the width and the height of the matrix respectively.

```
__kernel void editdistance(__global char* string1,
     __global char* string2, __global int* matrix,
     __global int* costs, __global int* dimensions,
     __global int* out) {

   int wid = get_global_id(0);

   int y = (dimensions[1] - 1)/ 2 - wid + 1;
   int x = -(dimensions[1] - 1)/ 2 + wid + 1;
```



		s	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
s	1								
u	2								
n	3								
d	4								
a	5								
y	6								

Figure 4. Starting positions of the work-items

This snippet contains the start of the main loop of the kernel. It computes the edit distance for the current cell of each work-item as described in section 5.

```
for (int i = 0; i < (dimensions[0] + dimensions[1] -
     1); i++) {
    if (x > 0 && x < dimensions[0] && y > 0 && y <
         dimensions[1]) {
        if (string1[x - 1] == string2[y - 1]) {
            matrix[to1D(x, y, dimensions[0])] =
                 matrix[to1D(x-1,y-1, dimensions[0])];
        } else {
```

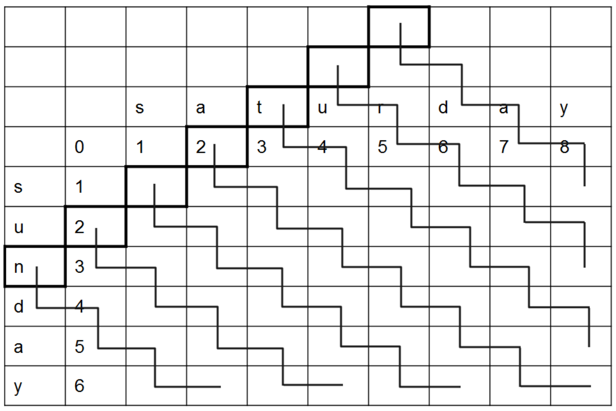


Figure 5. Division of labor between work-items

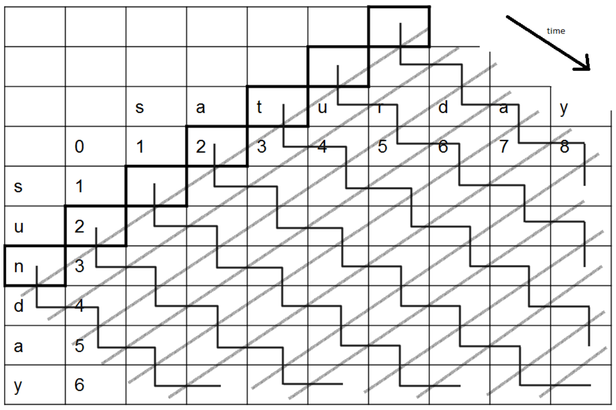


Figure 6. Division of labor between work-items over time, the grey lines indicate barriers

```
            matrix[to1D(x, y, dimensions[0])] =
                minimum(matrix[to1D(x-1,y,
                dimensions[0])] + costs[0],
                matrix[to1D(x,y-1, dimensions[0])] +
                costs[1], matrix[to1D(x-1,y-1,
                dimensions[0])] + costs[2]);
        }
    }
```

Finally, this snippet contains the end of the main loop, where x and y are updated. There is also a barrier here that forces work-items to wait until all work-items have reached the barrier. After the main loop, a single work-item returns the value in the lower right cell of the matrix, where the final result is stored.

```
    if (i % 2) {
        x++;
    } else {
        y++;
    }

    barrier(CLK_LOCAL_MEM_FENCE);
}

if (wid == 0) {
    out[0] = matrix[to1D(dimensions[0] - 1,
            dimensions[1] - 1, dimensions[0])];
}
```

6. SPECIFICATION

The program described in section 5.1 is specified using annotations. These describe the behaviour of the kernel as well as the permissions each work-item has to shared memory locations. There are two types of annotation used to specify the program: kernel behaviour specification, which describes the behaviour of the kernel, and shared memory location permission specification, which describes which work-items have read or write permissions to shared memory locations. The specification of a GPU program is typically divided into 3 levels:

- kernel level specs: describes all permissions transferred from the host to the kernel and the behaviour of the kernel as a whole.
- work-group level specs: describes how the permissions are distributed between different work-groups and the behaviour of the individual work-groups.
- work-item level specs: describes how the permissions are distributed among the work-items within the work-groups and the behaviour of the individual work-items.

Besides the different levels of specification, there are also barrier specficiations. These describe which permissions each work-item has after passing the barrier.

The complete program with specification can be found in the appendix, snippets are discussed below.

6.1 Specification of kernel behaviour

The main function has annotations that describe the behaviour of the kernel, work-group and individual work-items. The specification language used here is based on JML (Java Modeling Language). For the specificiation of this program there is no kernel specification as it is the same as the work-group specification because the program is restricted to only run on one work-group. The reason for this is that barriers can only synchronize work-items within the same work-group and synchronization between all work-items is necessary. The work-group specification has certain requirements about the parameters (mostly about the size of arrays) and ensures that every cell in the matrix meets certain conditions.

```
/*@
   (Work-group level specs)
    requires sizeof(costs) == (sizeof(cl_int) * 3) &&
         dimensions[0] > 0 && dimensions[1] > 0 &&
         sizeof(matrix) ==
         sizeof(dimensions[0])*sizeof(dimensions[1])
    requires (\forall int i ; i >= 0 ^ i < 3 =>
         costs[i] > 0);

    ensures out[0] >= 0
    ensures (\forall int x,y ; x > 0 ^ x <
         dimensions[0] ^ y > 0 ^ y < dimensions[1] ==>
         matrix[to1D(x,y, dimensions[0])] ==
         matrix[to1D(x - 1, y - 1, dimensions[0])] v
         matrix[to1D(x,y, dimensions[0])] ==
         minimum(matrix[to1D(x-1,y, dimensions[0])] +
         costs[0], matrix[to1D(x,y-1, dimensions[0])]
         + costs[1], matrix[to1D(x-1,y-1,
         dimensions[0])] + costs[2]));
 */
__kernel void editdistance(__global char* string1,
     __global char* string2, __global int* matrix,
     __global int* costs, __global int* dimensions,
     __global int* out) {
```

This snippet contains the loop invariants. These describe the behaviour of x and y with respect to the wid and loop iterator. Because c truncates towards zero upon integer division the fact that x and y increase alternatingly can be described by adding (i + 1) / 2 to the initial y position and i / 2 to the initial x position.

```
int y = (dimensions[1] - 1)/ 2 - wid + 1;
int x = -(dimensions[1] - 1)/ 2 + wid + 1;
for (int i = 0; i < (dimensions[0] + dimensions[1]
    - 1); i++) {
   //@loopinvariant y = (dimensions[1] - 1) / 2 -
       wid + 1 + (i + 1) / 2
   //@loopinvariant x = -(dimensions[1] - 1) / 2 +
       wid + 1 + i / 2

   ...

   if (i % 2) {
      x++;
   } else {
      y++;
   }
```

The specification ensures that D(x,y) = D(x - 1, y - 1) or that D(x,y) = minimum(D(k - 1, l) + C_i, D(k, l - 1) + C_d and D(k - 1, l - 1) + C_t). Essentially it ensures that the values in all the cells are computed as discussed in section 5.

```
   if (x > 0 && x < dimensions[0] && y > 0 && y <
       dimensions[1]) {
      if (string1[x - 1] == string2[y - 1]) {
         matrix[to1D(x, y, dimensions[0])] =
             matrix[to1D(x-1,y-1, dimensions[0])];
      } else {
         matrix[to1D(x, y, dimensions[0])] =
            minimum(matrix[to1D(x-1,y,
                dimensions[0])] + costs[0],
                matrix[to1D(x,y-1, dimensions[0])]
                + costs[1], matrix[to1D(x-1,y-1,
                dimensions[0])] + costs[2]);
      }
   }
   //@ensures matrix[to1D(x,y, dimensions[0])] ==
       matrix[to1D(x - 1, y - 1, dimensions[0])] v
       matrix[to1D(x,y, dimensions[0])] ==
       minimum(matrix[to1D(x-1,y, dimensions[0])]
       + costs[0], matrix[to1D(x,y-1,
       dimensions[0])] + costs[1],
       matrix[to1D(x-1,y-1, dimensions[0])] +
       costs[2]);
```

6.2 Specification of shared memory location permissions

To prove that a program does not have data races we need annotations to describe which permissions the work-items should have at different places in the kernel. Permissions are denoted by $perm(location, amount)$. The amount needed for a write permission is 100, for a read only permission it can be any amount under 100 or p for partial. The tool will be able to deduce the exact value of a read permission, so the programmer does not need to specify it.

The specification of shared memory location permissions consists of three important parts: the work-group level specification, the work-item level specification and the barrier specification. The work-group level specification describes which permissions each workgroup receives before running the kernel and which permissions are returned after running the kernel. The work-item level specification describes which permissions each work-item receives before running the kernel and which permissions are returned after running the kernel. The barrier specification describes all permissions each work-item receives at the barrier. Any permissions owned by a work-item before the barrier that are not mentioned in the barrier specification are automatically returned. The kernel level specification will also be automatically derived by the tool (the permissions received at the start are simply the sum of all permissions required by all work-items and the permissions that are returned are also the sum of all permissions returned by the work-items).

For the work-item and work-group level specification, requires means the permissions must be obtained before the work-item or work-group starts running and ensures means the permissions are returned after running. At the barrier there is only an ensures, the tool will derive which permissions are required before the work-item reaches the barrier.

Snippets of the code and specification are clarified below.

Because only one work-group is used, full permissions to all in- and outputs are required and ensured in the work-group level specification. On the work-item level specification partial permissions to all inputs are required. One of the work-items receives full permission to the cell at (1,1). The other work-items do not start at a location where computation is needed and will receive their write permissions later at the barrier in the main loop.

```
/*@
   (Work-group level specs)
   requires perm(string1, 100) * perm(string2, 100) *
       perm(matrix, 100) * perm(costs, 100) *
       perm(dimensions, 100) * perm(out, 100)
   requires \forall int x,y ; x > 0 ^ x <
       dimensions[0] ^ y > 0 ^ y < dimensions[1] ;
       perm(matrix[to1D(x, y, dimensions[0])], 100);
   ensures  perm(string1, 100) * perm(string2, 100) *
       perm(matrix, 100) * perm(costs, 100) *
       perm(dimensions, 100) * perm(out, 100)
   ensures  \forall int x,y ; x > 0 ^ x <
       dimensions[0] ^ y > 0 ^ y < dimensions[1] ;
       perm(matrix[to1D(x, y, dimensions[0])], 100);

   (Work-item level specs)
   requires perm(string1, p) * perm(string2, p) *
       perm(costs, p) * perm(dimensions, p)
   requires wid == (dimensions[1] - 1)/2 ==>
       perm(matrix[to1D(1,1, dimensions[0])], 100) *
       perm(matrix[to1D(0,1, dimensions[0])], p) *
       perm(matrix[to1D(1,0, dimensions[0])], p) *
       perm(matrix[to1D(0,0, dimensions[0])], p)
   ensures wid == 0 ==> perm(output, 100)
   ensures perm(string1, p) * perm(string2, p) *
       perm(costs, p) * perm(dimensions, p)
*/

__kernel void editdistance(__global char* string1,
     __global char* string2, __global int* matrix,
     __global int* costs, __global int* dimensions,
     __global int* out) {
```

At the barrier each work-item receives permission to the next cell (according to the updated x and y)

```
   barrier(CLK_LOCAL_MEM_FENCE);

   /*@
   x > 1 ^ y > 1 ==> {
        perm(matrix, p)
        perm(matrix[to1D(x - 1, y, dimensions[0])], p)
        * perm(matrix[to1D(x, y - 1, dimensions[0])],
            p)
        * perm(matrix[to1D(x - 1, y - 1,
            dimensions[0])], p)
        * perm(matrix[to1D(x, y, dimensions[0])], 100)
    }
    wid = 0 ^ i = dimensions[0] + dimensions[1] - 2
        ==> {
        perm(output, 100)
    }
    */
}
```

7. VERIFICATION

The goal is to eventually use VerCors to automatically verify OpenCL kernels by evaluating the program step by step (keeping track of all the conditions that hold along the way). Because this is very tedious work only a rough outline of the proof will be presented.

The input values (besides the cells in the matrix) will never cause any conflicts because not a single work-item ever receives write permission to any of them, the input is only read by the work-items. The output should not cause any issues either, as only one work-item writes to the output at the end of the kernel.

The specification of the permissions to the cells in the matrix are a little trickier (see the snippet above). Assuming the program behaves as described in section 5 and more specifically as in figure 4 and 5, we know that at the barrier all work-items are positioned along the same diagonal. This means that, since each work-item only receives write access to the cell at (x,y), no two work-items should have write access to the same cell in the matrix. Furthermore, work-items receive partial permissions to the cells located at (x - 1, y), (x, y - 1) and (x, y - 1). None of these are on the same diagonal as (x,y), so there are no cells of which a partial permission as well as a full permission has been given out to the work-items. This means that the program should be data race free and VerCors tool should be able to verify this program.

8. CONCLUSIONS AND FUTURE WORK

An OpenCL program that solves the edit distance problem was succesfully implemented. Both the behaviour and shared memory location permissions were specified. The work in this paper could be used as a case study for the further development of the VerCors tool set. In theory, VerCors will be able to verify the program and specification presented in this paper as correct, although minor syntax changes may be necessary.

8.1 Future work on the algorithm

Certain optimizations for the algorithm are possible. A minor optimization is not to calculate the first row and column on the host but to let the kernel take care of this. Another optimization is to redefine the to1D() function to change how two dimensional coordinates are translated to an array index. The kernel would require less memory fetches if the matrix is stored as a sequence of diagonals instead of a sequence of rows.

Because barriers are used (and are necessary), the kernel can only run on one work-group (barriers only synchronize within a single work-group). This means that when the length of the diagonals of the matrix exceeds the maximum work-group size the kernel can no longer compute entire diagonals at once, even though other GPU cores are idle. A possible solution is to divide the matrix into a certain amount of submatrices. The amount of submatrices needed depends on the max work-group size and the original matrix size. These submatrices can each be calculated by one work-group. Similar to how edit distance is solved in a matrix, the work-groups need to wait for each other to finish before they can start computing. An example is shown in figure 7. In the figure, three out of four quadrants are solved (the green/gray ones). The downside of this solution is that it is still not always possible to compute entire diagonals at a time. For example, during the last iteration of the active work-groups only the lower right corners of the second and third quadrants were computed (instead of the entire diagonal). In this case, if the max work-group size is three, the original solution would likely be more efficient. For larger matrices (where many diagonals are a lot larger than the maximum work-group size), it is likely worthwhile to split the matrix into submatrices so multiple work-groups can be used.

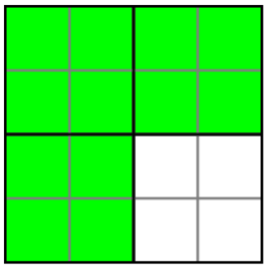


Figure 7. Partially solved matrix, divided into quadrants

9. REFERENCES

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The vercors project: setting up basecamp. In K. Claessen and N. Swamy, editors, *PLPV*, pages 71–82. ACM, 2012.

[3] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPUkernels. *SIGPLAN Not.*, 47(10):113–132, Oct. 2012.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. *SIGPLAN Not.*, 40(1):259–270, Jan. 2005.

[5] R. Farivar, H. Kharbanda, S. Venkataraman, and R. Campbell. An algorithm for fast edit distance computation on GPUs. 2012.

[6] M. Huisman and M. Mihelcic. Specification and verification of GPGPU programs using permission-based separation logic. Technical Report TR-CTIT-13-12, Centre for Telematics and Information Technology, University of Twente, Enschede, March 2013.

[7] Khronos Group. *The OpenCL Specification*, Sept. 2010.

[8] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, feb 1966. Doklady Akademii Nauk SSSR, V163 No4 845-848 1965.

[9] G. Li and G. Gopalakrishnan. Parameterized verification of GPU kernel programs. In *IPDPS Workshops*, pages 2450–2459. IEEE Computer Society, 2012.

[10] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[11] Wikipedia. Levenshtein distance — wikipedia, the free encyclopedia, 2013. [Online; accessed 16-October-2013].

10. APPENDIX

10.1 Complete kernel implementation & specification

```
/*@
   (Kernel_level specs)
   requires perm(string1, 100) * perm(string2, 100) *
        perm(matrix, 100) * perm(costs, 100) *
        perm(dimensions, 100) * perm(out, 100)
   requires (\forall int x,y ; x >= 0 ^ x <
        dimensions[0] ^ y >= 0 ^ y < dimensions[1] ;
        perm(matrix[to1D(x, y, dimensions[0])], 100)):

   requires sizeof(costs) == (sizeof(cl_int) * 3) &&
        dimensions[0] > 0 && dimensions[1] > 0 &&
        sizeof(matrix) ==
        sizeof(dimensions[0])*sizeof(dimensions[1])
   requires (\forall int i ; i >= 0 ^ i < 3 =>
        costs[i] > 0);

   ensures  perm(string1, 100) * perm(string2, 100) *
        perm(matrix, 100) * perm(costs, 100) *
        perm(dimensions, 100) * perm(out, 100)
   ensures  (\forall int x,y ; x > 0 ^ x <
        dimensions[0] ^ y > 0 ^ y < dimensions[1] ;
        perm(matrix[to1D(x, y, dimensions[0])], 100));

   ensures out >= 0
   ensures (\forall int x,y ; x > 0 ^ x <
        dimensions[0] ^ y > 0 ^ y < dimensions[1] ==>
        matrix[to1D(x,y, dimensions[0])] ==
        matrix[to1D(x - 1, y - 1, dimensions[0])] v
        matrix[to1D(x,y, dimensions[0])] ==
        minimum(matrix[to1D(x-1,y, dimensions[0])] +
        costs[0], matrix[to1D(x,y-1, dimensions[0])]
        + costs[1], matrix[to1D(x-1,y-1,
        dimensions[0])] + costs[2]));

   (work-item level specs)
   requires perm(string1, p) * perm(string2, p) *
        perm(costs, p) * perm(dimensions, p)
   requires wid == (dimensions[1] - 1)/2 ==>
        perm(matrix[to1D(1,1, dimensions[0])], 100) *
        perm(matrix[to1D(0,1, dimensions[0])], p) *
        perm(matrix[to1D(1,0, dimensions[0])], p) *
        perm(matrix[to1D(0,0, dimensions[0])], p)
   ensures wid == 0 ==>
        {perm(matrix[to1D(dimensions[0,0,
        dimensions[0])], 100)}
   ensures perm(string1, p) * perm(string2, p) *
        perm(costs, p) * perm(dimensions, p)
*/
__kernel void editdistance(__global char* string1,
    __global char* string2, __global int* matrix,
    __global int* costs, __global int* dimensions,
    __global int* out) {
   int wid = get_global_id(0);

   int y = (dimensions[1] - 1)/ 2 - wid + 1;
   int x = -(dimensions[1] - 1)/ 2 + wid + 1;

   for (int i = 0; i < (dimensions[0] + dimensions[1]
      - 1); i++) {
      //@loopinvariant y = (dimensions[1] - 1) / 2 -
          wid + 1 + (i + 1) / 2
      //@loopinvariant x = -(dimensions[1] - 1) / 2 +
          wid + 1 + i / 2

      if (x > 0 && x < dimensions[0] && y > 0 && y <
          dimensions[1]) {
         if (string1[x - 1] == string2[y - 1]) {
           matrix[to1D(x, y, dimensions[0])] =
                matrix[to1D(x-1,y-1, dimensions[0])];
         } else {
           matrix[to1D(x, y, dimensions[0])] =
              minimum(matrix[to1D(x-1,y,
                   dimensions[0])] + costs[0],
                   matrix[to1D(x,y-1, dimensions[0])]
                   + costs[1], matrix[to1D(x-1,y-1,
                   dimensions[0])] + costs[2]);
         }
      }
      //@ensures matrix[to1D(x,y, dimensions[0])] ==
          matrix[to1D(x - 1, y - 1, dimensions[0])] v
          matrix[to1D(x,y, dimensions[0])] ==
          minimum(matrix[to1D(x-1,y, dimensions[0])]
          + costs[0], matrix[to1D(x,y-1,
          dimensions[0])] + costs[1],
          matrix[to1D(x-1,y-1, dimensions[0])] +
          costs[2]);

      int oldX = x;
      int oldY = y;

      if (i % 2) {
         x++;
      } else {
         y++;
      }

      barrier(CLK_LOCAL_MEM_FENCE);

      /*@
      x > 1 ^ y > 1 ==> {
         perm(matrix, p)
         perm(matrix[to1D(x - 1, y, dimensions[0])], p)
         * perm(matrix[to1D(x, y - 1, dimensions[0])],
             p)
         * perm(matrix[to1D(x - 1, y - 1,
             dimensions[0])], p)
         * perm(matrix[to1D(x, y, dimensions[0])], 100)
      }
      wid = 0 ^ i = dimensions[0] + dimensions[1] - 2
          ==> {
         perm(output, 100)
      }
      */
   }


   if (wid == 0) {
      out[0] = matrix[to1D(dimensions[0] - 1,
          dimensions[1] - 1, dimensions[0])];
   }
}
```