III. International Conference on Electrical, Computer and Energy Technologies (ICECET 2023)
16-17 November 2023, Cape Town-South Africa

A Parallel Connected Region Algorithm Optimized for CPU-GPU Heterogeneous Platforms

David Troendle
Computer and Information Science
The University of Mississippi
University, MS USA
david@cs.olemiss.edu

Byunghyun Jang
Computer and Information Science
The University of Mississippi
University, MS USA
bjang@cs.olemiss.edu

2023 International Conference on Electrical, Computer and Energy Technologies (ICECET) | 979-8-3503-2781-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICECET58911.2023.10389550

***Abstract*—In recent years, GPU has become an important component of the system by accelerating applications with data parallelism. While traditional discrete GPU cooperates with CPU over I/O bus, more recent processor design trends feature fine-grained synchronization and inexpensive data sharing between CPU and GPU processors by tightly coupling them at a cache level. However, designing algorithms that effectively exploit these new architectural features has proved challenging.**

In this paper we present a novel parallel algorithm called PARAFILL that identifies connected regions of arbitrary topologies in an image very efficiently on CPU-GPU heterogeneous platforms. PARAFILL is a greedy algorithm using dynamic programming *memoization* techniques. Essential to its efficiency is a data structure called the *workload manager*, which dynamically manages the complexity of work in each pass. We evaluate the performance of PARAFILL across two different heterogeneous platforms: a traditional discrete GPU system where the CPU and GPU are connected through I/O bus, and a single-chip processor where the CPU and GPU are tightly coupled at a cache level. We also compare PARAFILL to a CPU-optimized connected region algorithm using applications with different parallelism to quantify its scalability. Our experimental results show that minimizing GPU overhead and managing pass-by-pass workload are especially important for sparsely parallel applications. Tightly coupled GPUs, although less powerful than discrete GPUs, compete well in sparsely parallel applications because of their reduced overhead. Finally, we present two applications of PARAFILL: traditional flood filling and the absorption of unconnected interior regions into a containing connected region.

***Index Terms*—GPU Programming, Parallel Algorithms, Flood Fill, Connected Regions**

I. INTRODUCTION

Identifying the connected regions of an image or other forms of data is a basic yet very important task that finds many applications in the fields of computer graphics and computer vision [1, 2]. For example, it is commonly used in paint programs (e.g., Adobe Photoshop), image segmentation, or solving mazes. Although there are several well-known algorithms for this task [3, 4], they suffer from high computation demands and can be very slow for certain topologies, especially large topologies.

General Purpose computation on GPUs (GPGPU) provides opportunity for significant acceleration of data and compute intensive applications [5]. Traditional GPGPU platforms are powered by discrete GPUs connected through I/O bus. Although many algorithms and applications have been successfully accelerated using GPUs, there remain many unaccelerated algorithms and applications due to sparse parallelism and GPU-related overhead. To address this issue and provide other benefits such as low design cost and fast on-chip interconnection, recent industry trends have merged CPUs with GPUs on a single processor die. Such CPU-GPU merged processors are prevalent in the market as of the writing of this paper (AMD APUs [6] and Intel Sandy Bridge/Ivy Bridge/Haswell [7]).

Regardless of hardware configurations (i.e., traditional discrete GPUs and relatively new tightly coupled processors), designing and optimizing workloads (e.g., algorithms or applications) that fully exploit the hardware features is a challenging task. Workloads must be carefully divided into two different kinds of tasks, one of which is executed on a task parallel CPU, while the other is executed on a data-parallel GPU. Techniques such as simple workload distribution/balancing are not enough to take full advantage of computing power that hardware has to offer. For small problem sizes, a GPU may not be the best choice.

In this paper, we introduce a new parallel algorithm, PARAFILL, which efficiently identifies connected regions in images or other data on CPU-GPU heterogeneous platforms. PARAFILL is a greedy algorithm that uses dynamic programming's *memoization* technique. It takes an iterative approach, processing the work for the current pass while discovering new work for the next pass. A data structure called *workload manager* is a key to the efficiency of the algorithm by managing the complexity of each pass. Applications rich in intrinsic data parallelism typically process against all data in one pass. Vector addition and matrix multiplication are good examples. In application where there is only sparse parallelism that must be discovered over many passes, processing against all data in each pass is manifestly inefficient. We demonstrate the overhead of tracking and processing just the parallel opportunities of the current pass can be less expensive than processing the entire problem geometry in each pass. In PARAFILL's case,

979-8-3503-2781-6/23/$31.00 ©2023 IEEE

managing workload requires using inefficient operations such as barriers and atomic operations.

Our contribution is a region connection algorithm for GPU-powered platforms. We openly discuss the problems we faced designing an algorithm for a sparsely parallel problem. We share the techniques used to address these problem in the hope others will attempt parallel designs for sparsely parallel problem and improve upon our techniques.

Current hardware requires relatively large topologies for effective use of the algorithm. All parallel applications have a crossover point where the GPU becomes an effective part of solving the problem. Our results show that for small or complicated connected regions, the CPU has the advantage and the GPU should not be used. The primary reason is the significant overhead associated with using a GPU. As the size of the region increases the advantage trends to the GPUs (i.e., integrated GPU and discrete GPU). Because parallelism is discovered over a series of passes, most passes do not have sufficient parallelism to fully occupy a high-end GPU. There simply are not enough work groups for all the CUs. Consequently a relatively small APU (i.e., integrated GPU) was able to hold its own against high-end discrete GPUs thanks to mainly its efficient data sharing. In our real-world "Hand" image example, when 6 seed points were used, the APU was 39% faster than the optimized CPU algorithm while the discrete GPU was 33% faster. Given the high-end GPU used was several times more expensive than the APU, our results call into question the cost-effectiveness of a high-end discrete GPU for sparsely parallel applications.

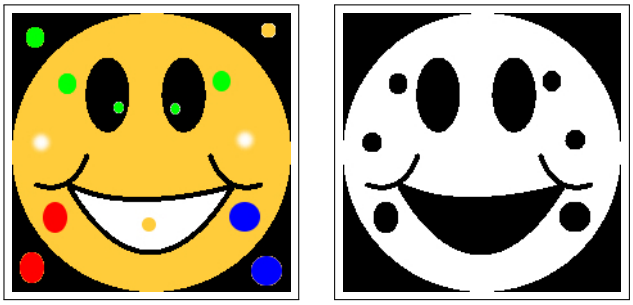
To thoroughly evaluate the efficiency of PARAFILL on heterogeneous processors, we developed several implementations of the algorithm. For the CPU we developed an optimized CPU-only connected region algorithm[1]. For the GPU we developed two variants with differing workload management techniques. Our experiments show that for this application tightly coupled heterogeneous processors compete well against the more powerful discrete GPU.

II. IDENTIFYING CONNECTED REGION

Identification of connected regions is an important problem in computer graphics and related fields. It is applicable to multiple dimension arrays. In this paper we limit our discussion to 2-D images but the technique presented is expandable to more dimensions and other types of data.

Connected region identification requires an adjacency criteria. For images, the most commonly used adjacency criteria is color. For instance Phung et al. [8] compare the effectiveness of several color spaces for the skin segmentation problem. Their criteria for adjacency is skin tone. For our benchmarking, we used the CIE-Lab XYZ color space based on the maximum tolerable distance from a reference color. We chose the XYZ color space because it more closely represents the way the typical human eye perceives color and intensity. Each channel has its own tolerance because it is common to allow more tolerance on the luminance (Y) channel.

Consider the smiley face show in Figure 1a. Using the dominating yellow as the reference color, and the center of the image as the starting pixel Figure 1b shows the resulting connected region.



(a) Sample image (b) Connected region

Fig. 1. Sample image and connected region.

Note that even though it is a connectable color, the yellow dot inside the mouth area is eliminated because it is surrounded by the white area of the mouth, which blocks connections. The other colors on the face (green, blue, white, and black) are not within the tolerance used and are thus unconnected.

For our experiments, the input is (1) an image, (2) a reference color and tolerance, and (3) a vector of starting pixels. The output is a binary image, where white pixels identify the connected region and the black pixels are the unconnected pixels.

III. THE PARAFILL ALGORITHM

To determine if a pixel is part of a connected region, we must first know if one of its neighbor is connected. This induces an order in which pixels must be checked for connectivity, and makes finding the connected region for an arbitrary topology difficult to parallelize. PARAFILL addresses this challenge by processing the region in a systematic fashion.

A. Design Challenges

As the software and hardware enabling heterogeneous computing evolve, the potential uses expand. Discrete GPUs are I/O devices – all commands and data have to be sent to and from the device, causing significant overhead. Overcoming this overhead requires significant data parallel acceleration on a GPU. To be effective, the GPU execution time plus overhead time have to be less than performing the task on the CPU.

There is an additional overhead consideration for iterative algorithms such as PARAFILL. If there are kernel launches in each iteration, then the kernel launch overhead costs are multiplied by the number of iterations. Reduction of overhead means less data parallelism is required for effective use of a GPU.

[1]The connected region and *flood filling* problems are closely related but different problems. Flood filling identifies the connected region *and* changes the color of the connected region, while PARAFILL focuses solely on the identification of the connected region. Absorption of unconnected regions wholly contained in a connected region highlights the difference in the two approaches.

Data transfers via I/O are the most significant overhead component. To address this, vendors integrated the GPU into the processor die and shared physical memory with the CPU cores via a last level shared cache. The effect is that the CPU and GPU address the same physical memory, possibly through different virtual addresses. This transforms the data I/O transfer problem into a far less expensive virtual address management problem. Placing the GPU on the processor die, however, limits the number of GPU compute units (CUs). Typically integrated GPUs (e.g., AMD APU, Intel Sandy Bridge/Ivy Bridge/Haswell) have far fewer CUs than discrete GPUs. For applications with limited data parallelism, this is a good trade-off. For large massively data parallel applications, discrete GPUs may be better suited.

In PARAFILL each pass operates on a relatively small subset of the image. Thus the index range for each pass need not be all pixels in the image. PARAFILL uses a workload manager data structure to memoize the image subset where parallelism is available, and sets the index range accordingly. The workload manager is simply a queue of spans needing processing. The size of the queue is the complexity of the pass. As long as the workload manager has queued spans, those spans are processed in parallel in the next pass. To minimize overhead, PARAFILL keeps all data on the GPU and transfers only necessary data. Only a single *scalar integer* is transferred to and from the GPU at the end of each pass. There is a delay from when a kernel is launched to when it actually begins execution. This is called kernel launch overhead, and is another significant overhead factor. Section IV-B discusses the mitigation of this overhead in detail.

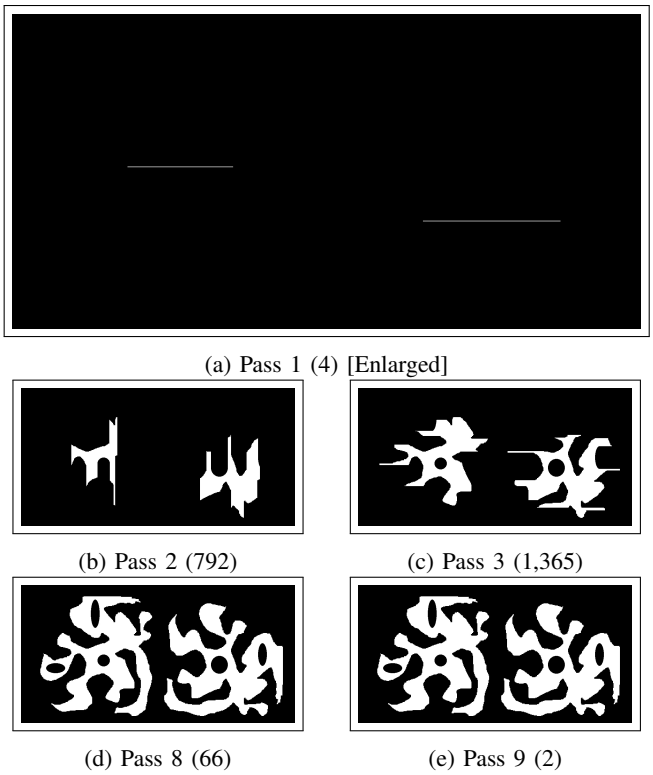
B. Algorithm Description

PARAFILL is a greedy algorithm using dynamic programming memoization techniques. A workload management queue memoizes work discovered for the next pass and helps keep pass complexity low. Given an image, the starting pixel location(s) and a color reference/tolerance, PARAFILL iteratively finds all pixels connected to the starting pixels. The pseudo code for the CPU version of our proposed algorithm is given in Algorithm 1.

Algorithm 1 PARAFILL algorithm

```
1:  function ParaFill(Image, StartingPixels, Color, Tolerance)
2:      MoreWork ← Initialize
3:      Direction ← Horizontal
4:      while MoreWork do
5:          if Direction == Horizontal then
6:              MoreWork ← HorizontalPass
7:          else
8:              MoreWork ← VerticalPass
9:          end if
10:         Direction ← OtherDirection(Direction)
11:     end while
12:     return ConnectedRegion
13: end function
```

To see the big picture on how the algorithm works, we first show sample results. Figure 2 shows the output of select passes for the jigsaw image. The pass-by-pass parallelism (i.e., the number of spans processed in the pass) is shown in parenthesis after each pass number. For pass 1 there are 4 spans – a left and right horizontal span for each of the two pieces. The two pieces are simultaneously connected in 9 passes (iterations). Note that pass 1 is enlarged so the two initial horizontal spans become faintly visible.



(a) Pass 1 (4) [Enlarged]

(b) Pass 2 (792)

(c) Pass 3 (1,365)

(d) Pass 8 (66)

(e) Pass 9 (2)

Fig. 2. Jigsaw sample iterations.

The algorithm works as follows: *Initialize* (line 2) initializes the output image to black, indicating no connected pixels have been found. It then examines the starting pixel(s), queuing appropriate span entries in the workload manager. It then enters the main loop.

At the start of each pass the workload manager has the starting pixels for each span. *HorizontalPass* (line 6) spans horizontally until it reaches a boundary (see definition below). It marks each pixel along the span as connected, examining each pixel's neighbor above and below. Connectable neighbors above are queued for upward spans in the next pass while connectable neighbors below are queued for downward spans in the next pass.

VerticalPass (line 8) proceeds in a similar fashion except it spans vertically. It marks each pixel along the span as connected, examining each pixel's neighbor to the left and right. Connectable neighbors to the left are queued for left spans in the next pass while connectable neighbors to the right are queued for right spans in the next pass.

A *boundary* is defined as any one of the following:

- Spanning outside the image borders. This is detected using the input image geometry.
- Spanning to a pixel already marked as connected. This is detected using a memoization table of marked, connected pixels. After the last iteration, this memoization table contains the connected region(s).
- Spanning to a pixel that does not meet adjacency criteria. This is detected using the reference color and tolerance.

C. Optimizations

Avoiding redundancy: There is an efficiency issue with the parallel version of Algorithm 1. Recall that in the OpenCL programming model you can make no assumption regarding processing order. We now examine a problematic processing order. Consider any three adjacent vertical spans. Figure 3 depicts processing the 4^{th} pixel in three adjacent spans. The black stars represent marked, connected pixels. The outlined white star represents an unmarked but connectable pixel.

Assume the center span processes last, which delays marking the pixel as connected. The leftmost of the three spans looks to its right neighbor and discovers an as yet unconnected pixel and queues it for processing in the next pass. Likewise the rightmost span looks to its left neighbor and discovers the same as yet unconnected pixel and redundantly queues it for processing in the next pass.

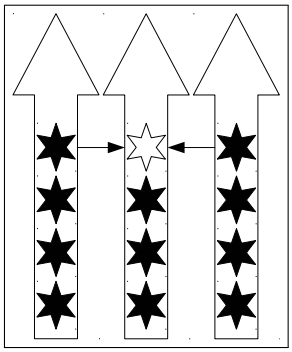


Fig. 3. Redundant span queuing

Eventually the center span marks the pixel connected, effectively rendering the queuing done by *both* the left and right spans redundant and inefficient. This is not a correctness issue because there is global synchronization between the passes and in the next pass the redundant span(s) are immediately stopped by the now-connected pixel. Recall that a connected pixel is a boundary and will stop a span.

Reducing redundant queuing: From the above problematic processing order we can see the most advantageous order is for all connections of a pass to be done before checking neighbors for adjacency.

The more pixels connected before checking their neighbors the fewer the redundantly queued spans.

We achieve this by breaking a pass into two phases – a "connecting" phase and a "neighbor checking" phase. In the first phase connectable pixels along the span are marked as connected and the path memoized. Next synchronization is required so that all pixels along the spans of the current pass are connected before beginning the second "neighbor checking" phase. Global synchronization will ensure no redundant pixels are queued, but this requires two kernel launches, which incurs overhead. We call the global synchronization approach the 2-kernel approach.

Alternatively we can achieve most of the desired effect by synchronizing within each work group using a barrier to separate the phases. This significantly reduces the chances of redundant queuing while using only one kernel.

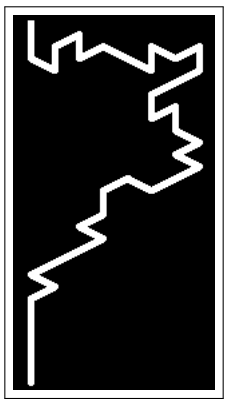
In the second phase, the neighbors of the pixels along the memoized spans are checked for adjacency. Adjacent neighbors still unconnected that meet the connection criteria are queued for parallel processing in the next pass.

It should be noted the 1-kernel approach, while correct, is not a deterministic. The 2-kernel approach is deterministic. There are many factors that affect the exact order of work group execution, which in turn affects redundant queuing. Further, if the pass that connected the final pixels should queue redundant work, an extra pass (connecting no pixels) is required to clear the work queue. The effect ends with the extra pass because it queues no further spans.

Multiple starting points: Note that this approach allows for multiple staring points. *Initialize* simply queues a span for each starting point. If the starting points are redundant or in the same region it causes no problem. More interestingly, the starting pixels can be in different regions. When this happens, PARAFILL connects the regions in parallel. See Figure 2 for an example. The CPU-only version of the algorithm must connect each region serially.

IV. PERFORMANCE EVALUATION AND ANALYSIS

We evaluated PARAFILL's performance across a wide spectrum of inputs and starting conditions. We benchmarked five images (including a real-world image) with different characteristics. They are listed and described in Table III.



(a) Path (Scale=0.5x)

(b) Hand (Scale=0.045x)

Fig. 4. Sample benchmark images.

Table IV summarizes the specifications of hardware used in our experiments. The information was obtained from the

TABLE I. Execution time analysis (seconds for 100 iterations).

		Trinity APU			Tahiti GPU		
	CPU	Complexity Management			Complexity Management		
Image	Connected	None	1-Kernel	2-Kernel	None	1-Kernel	2-Kernel
Path	**0.00845106**	1.24809000	0.08736730	0.08058060	0.74104500	0.41972500	0.48188700
Smile	**0.02859232**	0.25492800	0.06839640	0.07897180	0.25775100	0.21326400	0.26295700
Jigsaw	**0.23433990**	2.86139000	0.45200000	0.66194800	2.38367000	0.51604900	0.63093400
Square	5.15282480	11.78020000	5.06227000	7.97068000	1.67637000	**0.68106600**	0.80807600
Hand1SP	6.40684200	54.86030000	4.73156000	6.61682000	43.90250000	**3.90011000**	4.41799000
Hand6SP	6.35882400	57.52500000	**3.84617000**	5.32592000	44.08930000	4.02787000	5.08939000

TABLE II. Execution time analysis with overhead (seconds for 100 iterations).

		Trinity APU			Tahiti GPU		
	CPU	Complexity Management			Complexity Management		
Image	Connected	None	1-Kernel	2-Kernel	None	1-Kernel	2-Kernel
Path	**0.00845106**	1.610280000	0.21592400	0.21173300	1.889510000	0.95615600	1.09142000
Smile	**0.02859232**	0.338008000	0.09249990	0.10669500	0.510379000	0.38949900	0.46442900
Jigsaw	**0.23433990**	2.998790000	0.49233900	0.72920500	2.766530000	0.84761200	0.99653200
Square	5.15282480	11.909400009	5.08344000	8.00240000	1.778680000	**0.77179500**	0.91180600
Hand1SP	6.40684200	55.07060000	4.77660000	6.71834000	44.36160000	**4.17927000**	4.74768000
Hand6SP	6.35882400	57.74810000	**3.88922000**	5.41485000	44.56790000	4.28913000	5.39852000

TABLE III. Benchmark images.

Image	Geometry	Rationale
Path	151×281	Figure 4a. This is a small image with a jagged path. The jaggedness requires many PARAFILL iterations.
Smiley	217×217	Figure 1a. This is a small image with many challenging features, especially for the absorption of internal unconnected regions.
Jigsaw	1K×512	This is a medium sized image. It contains several connectable regions with several challenging features. It is used to demonstrate simultaneously regions filling. Whenever this image is used the two larger regions are connected.
Square	1K×1K	Not shown. This is a large, square, single colored image. All pixels are connected. While not a typical connected region topology, we include it to examine PARAFILL's performance in a richly parallel environment.
Hand	4,128×3,096	Figure 4b. This is a very large real-world image with many features, included jagged boundaries. We evaluated performance with a single starting point as well as six starting points.

TABLE IV. Specifications of experiment platforms.

	CPU	Discrete GPU	Integrated GPU (APU)
Vendor	AMD	AMD	AMD
Device	A10-5800K	HD 7970	A10-5800K
CUs/Cores	4	32	4
Max Freq.	3.8GHz	925MHz	800MHz
Max WG Size	N/A	256	1,024
Unified Memory	N/A	No	Yes

"clinfo" utility. We used A10-5800K (codenamed "Trinity") APU. To represent a high-end discrete GPU, we used HD 7970 (codenamed "Tahiti") GPU which is based on AMD's Graphics Core Next (GCN) microarchitecture.

A. Parallelism

We measure parallelism for a pass by the number of spans in a pass. There will be one work item for each span. The true parallelism for a pass is the number of non-redundant spans in a pass given by the 2-kernel approach.

There are three factors affecting performance of PARAFILL. The first factor is parallelism intrinsic to the data. The topology of the connected region affects the parallelism. For instance, two images of the same size with a different connected region topologies will offer different parallelism opportunities. For connected regions, the maximum parallelism is $max(Height, Width)$ (the pixels in the perpendicular direction are spanned), which is far less than the total number of pixels in the image. (The "Square" image illustrates this.) The second factor is the algorithm's effectiveness in discovering the intrinsic parallelism of the data. The final factor is the GPU hardware. While speed is always important for hardware, other factors such as overhead are important and must be considered.

Table V details the pass-by-pass parallelism discovered using 1) all pixels without workload management, 2) the one-kernel approach and 3) the two-kernel approach. Since the one-kernel approach is not deterministic, results for the APU and GPU can differ and are therefore each detailed in the table.

TABLE V. Parallelism for smiley image.

	No Workload	1-Kernel		2-Kernel
Pass	Management	APU	GPU	(Actual)
1	47,089	2	2	2
2	47,089	424	424	424
3	47,089	776	693	693
4	47,089	197	136	123
5	47,089	189	131	131
6	47,089	92	98	67
7	N/A	6	N/A	N/A

B. GPU Kernel Launch Overhead

As the level of data parallelization decreases, GPU launch overhead becomes an increasingly significant issue. This is especially true for iterative algorithms because each pass incurs that overhead. There are three phases involved in the execution of an OpenCL kernel: queuing, submission and execution.

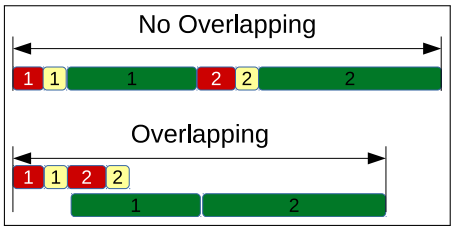


Fig. 5. Overlapping kernel launch overhead.

The top part of Figure 5 shows the launch of two kernels separated by a *clFinish*. The red box represents queue time, the yellow submit time and the green execution time. Notice that kernel 2's execution is delayed by its launch overhead time. The bottom part of Figure 5 shows how time can be saved with overlapping. Both kernels are launched but there is no *clFinish*. OpenCL events are used to delay the execution of kernel 2 until kernel 1 completes, but kernel 2's launch overhead time overlaps kernel 1's execution time. The effect is that when kernel 1 ends, kernel 2 starts almost immediately. Table VI shows typical results of the effect of overlapping the queue time of the second kernel with the execution of the first kernel. Our experiments show that the APU queuing times are significantly smaller than the GPU's queuing times, and overlapping offered on modest improvement (except for the "Hand" image where there was modest cost). For the Tahiti GPU, the queuing time for the second kernel was strikingly lower. These results strongly suggest overlapped queuing is an important technique for reducing overhead for the Tahiti GPU. The Trinity's architecture is very different and does not significantly benefit from this technique (Table VI).

TABLE VI. Effect of overlapped queuing (100 iterations in seconds).

	APU		Tahiti GPU	
Image	1^{st} kernel	2^{nd} kernel	1^{st} kernel	2^{nd} kernel
Path	0.0312249	0.0131871	0.2278180	0.0023440
Smile	0.0051703	0.0048576	0.0861462	0.0003164
Jigsaw	0.0117529	0.0105311	0.1581100	0.0005232
Square	0.0027247	0.0019998	0.0340680	0.0001142
Hand1SP	0.0138149	0.0194019	0.1330140	0.0007069
Hand6SP	0.0122907	0.0380242	0.1285040	0.0007434

C. Total Execution Time

Table I details execution times for the two GPUs and three complexity models. The best time for each image is shown in bold. Where the GPU was not the fastest time, the best GPU time is highlighted gray. Table II shows the results with overhead included. The best times for each algorithm are shown in bold. Where the GPU was not the fastest time, the best GPU time is highlighted gray. Inclusion of overhead did not change the results.

In Tables I and II we notice the 1-kernel approach was uniformly faster than the 2-kernel approach, whenever the GPU was the best option. We see the CPU is better-suited for processing the three smaller images. However, for those images notice the APU outperformed the discrete GPU. As the parallelism increased the APU's becomes more effective.

The "Square" image is not a typical connected region topology, but was included to explore the effects of a richly parallel topology. The Tahiti GPU outperformed the optimized CPU algorithm by an impressive 6.7 times. The APU outperformed the optimized CPU algorithm by 1% – essentially a tie.

The result for the hand with 6 starting points is interesting. There the APU was faster than the Tahiti GPU. Notice the extra starting points increased parallelism in passes 2 and 3. This favored the Tahiti GPU, but passes 4-10 with less parallelism favored the APU. This gave the slight advantage to the APU as seen in the results. Considering the high-end Tahiti GPU is several times more expensive than the more modest APU, we question the cost-effectiveness of a high-end discrete GPU in sparsely parallel applications.

ACKNOWLEDGMENT

This research is supported by the National Science Foundation under Grant 1907838.

REFERENCES

[1] Y.-I. Ohta, T. Kanade, and T. Sakai, "Color information for region segmentation," *Computer graphics and image processing*, vol. 13, no. 3, pp. 222–241, 1980.

[2] K. E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, ser. AFIPS '68 (Spring). New York, NY, USA: ACM, 1968, pp. 307–314. [Online]. Available: http://doi.acm.org/10.1145/1468075.1468121

[3] R. Adams and L. Bischof, "Seeded region growing," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 16, no. 6, pp. 641–647, Jun 1994.

[4] E. Nosal, "Flood-fill algorithms used for passive acoustic detection and tracking," in *New Trends for Environmental Monitoring Using Passive Systems, 2008*, Oct 2008, pp. 1–5.

[5] NVIDIA. GPU Applications. http://www.nvidia.com/object/gpu-applications-domain.html.

[6] AMD. (2014) AMD Accelerated Processing Units (APUs). [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/apu

[7] Intel. (2014) Intel Core Processor Family. [Online]. Available: http://www.intel.com/content/www/us/en/processors/core/core-processor-family.html

[8] S. Phung, A. Bouzerdoum, and S. Chai, D., "Skin segmentation using color pixel classification: analysis and comparison," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 1, pp. 148–154, Jan 2005.

[9] HSA Foundation. (2014) Heterogeneous System Architecture (HSA) Foundation. [Online]. Available: http://hsafoundation.com/

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

[11] R. C. Gonzalez and R. E. Woods, *Digital Image Processing (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006.