HSF Reconstruction and Software Trigger Working Group

Towards Zero-Waste Computing with Performance Engineering

A Practical Example from Track Reconstruction

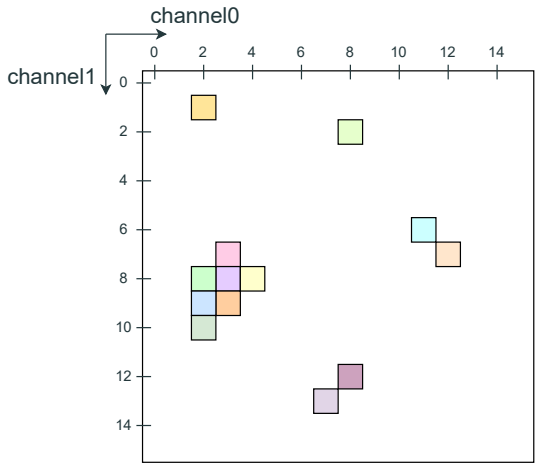
Stephen Nicholas Swatman[1,2], Ana-Lucia Varbanescu[3]

Wednesday, September 13th, 2023

[1]University of Amsterdam [2]CERN [3]University of Twente

Introduction – Clustering

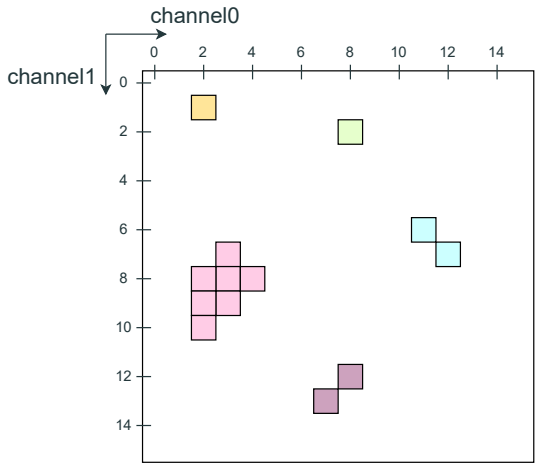
- Pixel detectors give **deposition** per pixel
- **One particle** may activate **many pixels**
- But no **cluster information** (let alone particle information) can be recorded
- Computing clusters is one of the first steps in reconstruction



Graphic by Uchendu Nwachukwu

Introduction – Clustering

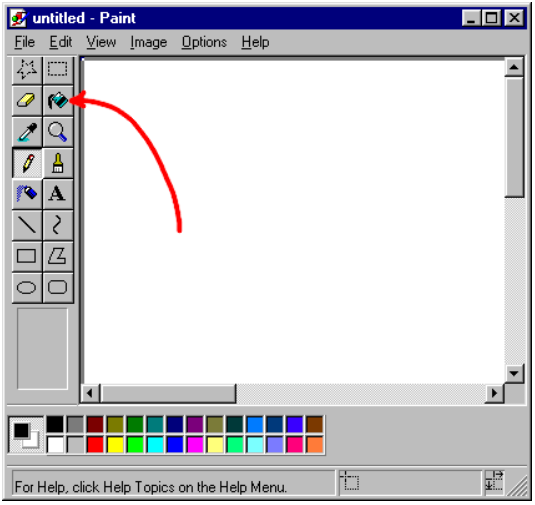
- Pixel detectors give **deposition** per pixel
- **One particle** may activate **many pixels**
- But no **cluster information** (let alone particle information) can be recorded
- Computing clusters is one of the first steps in reconstruction



Graphic by Uchendu Nwachukwu

Introduction – Connected Component Labelling

- In graph theory and computer vision, known as **connected component labelling (CCL)**
  - You might recognise its greatest hit: the Microsoft Paint Fill tool
- Usually applied to **graph** or **dense image** data
- **Vast body of knowledge** on algorithms across all sorts of devices



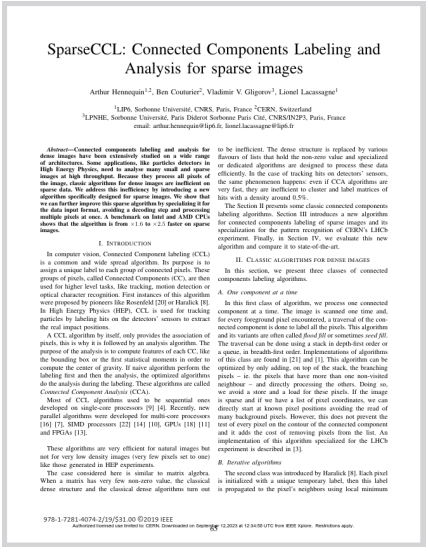
Zero Waste Computing – CCL for Sparse Data

- CCL problems in **HEP** are interesting because they are...
  - Extremely **sparse** (~2% N.Z.)
  - Across **disjoint images** (~2000 in ATLAS ID)
- Sparse problems are **less common** and there is **less work on algorithm design**
- **SparseCCL** was developed for HEP applications at CERN and Sorbonne University by A. Hennequin et al. (doi:10.1109/DASIP48288.2019.9049184)

Flavour	Positioning	Connection
Dense	Implicit	Implicit
Sparse	Explicit	Implicit
Graph	N.A.	Explicit

Zero Waste Computing – SparseCCL for CPU

- SparseCCL runs **sequentially** over an image
  - We'll assume that SparseCCL is efficient
- How can we now run this efficiently on **multi-core** systems?
  - In other words: *where is the parallelism*?



SparseCCL: Connected Components Labeling and Analysis for sparse images

Arthur Hennequin[1,2], Ben Couturier[2], Vladimir V. Gligorov[3], Lionel Lacassagne[1]

[1]LIP6, Sorbonne Université, CNRS, Paris, France [2]CERN, Switzerland
[3]LPNHE, Sorbonne Université, Paris Diderot Sorbonne Paris Cité, CNRS/IN2P3, Paris, France
email: arthur.hennequin@lip6.fr, lionel.lacassagne@lip6.fr

***Abstract*—Connected components labeling and analysis for dense images have been extensively studied on a wide range of architectures. Some applications, like particles detectors in High Energy Physics, need to analyse many small and sparse images at high throughput. Because they process all pixels of the image, classic algorithms for dense images are inefficient on sparse data. We address this inefficiency by introducing a new algorithm specifically designed for sparse images. We show that we can further improve this sparse algorithm by specializing it for the data input format, avoiding a decoding step and processing multiple pixels at once. A benchmark on Intel and AMD CPUs shows that the algorithm is from ×1.6 to ×2.5 faster on sparse images.**

I. Introduction

In computer vision, Connected Component labeling (CCL) is a common and wide spread algorithm. Its purpose is to assign a unique label to each group of connected pixels. These groups of pixels, called Connected Components (CC), are then used for higher level tasks, like tracking, motion detection or optical character recognition. First instances of this algorithm were proposed by pioneers like Rosenfeld [20] or Haralick [8]. In High Energy Physics (HEP), CCL is used for tracking particles by labeling hits on the detectors' sensors to extract the real impact positions.

A CCL algorithm by itself, only provides the association of pixels, this is why it is followed by an analysis algorithm. The purpose of the analysis is to compute features of each CC, like the bounding box or the first statistical moments in order to compute the center of gravity. If naive algorithm perform the labeling first and then the analysis, the optimized algorithms do the analysis during the labeling. These algorithms are called *Connected Component Analysis* (CCA).

Most of CCL algorithms used to be sequential ones developed on single-core processors [9] [4]. Recently, new parallel algorithms were developed for multi-core processors [16] [7], SIMD processors [22] [14] [10], GPUs [18] [11] and FPGAs [13].

These algorithms are very efficient for natural images but not for very low density images (very few pixels set to one) like those generated in HEP experiments.

The case considered here is similar to matrix algebra. When a matrix has very few non-zero value, the classical dense structure and the classical dense algorithms turn out to be inefficient. The dense structure is replaced by various flavours of lists that hold the non-zero value and specialized or dedicated algorithms are designed to process these data efficiently. In the case of tracking hits on detectors' sensors, the same phenomenon happens: even if CCA algorithms are very fast, they are inefficient to cluster and label matrices of hits with a density around 0.5%.

The Section II presents some classic connected components labeling algorithms. Section III introduces a new algorithm for connected components labeling of sparse images and its specialization for the pattern recognition of CERN's LHCb experiment. Finally, in Section IV, we evaluate this new algorithm and compare it to state-of-the-art.

II. Classic algorithms for dense images

In this section, we present three classes of connected components labeling algorithms.

A. One component at a time

In this first class of algorithm, we process one connected component at a time. The image is scanned one time and, for every foreground pixel encountered, a traversal of the connected component is done to label all the pixels. This algorithm and its variants are often called *flood fill* or sometimes *seed fill*. The traversal can be done using a stack in depth-first order or a queue, in breadth-first order. Implementations of algorithms of this class are found in [21] and [1]. This algorithm can be optimized by only adding, on top of the stack, the branching pixels – ie. the pixels that have more than one non-visited neighbour – and directly processing the others. Doing so, we avoid a store and a load for these pixels. If the image is sparse and if we have a list of pixel coordinates, we can directly start at known pixel positions avoiding the read of many background pixels. However, this does not prevent the test of every pixel on the contour of the connected component and it adds the cost of removing pixels from the list. An implementation of this algorithm specialized for the LHCb experiment is described in [3].

B. Iterative algorithms

The second class was introduced by Haralick [8]. Each pixel is initialized with a unique temporary label, then this label is propagated to the pixel's neighbors using local minimum

978-1-7281-4074-2/19/$31.00 ©2019 IEEE

Zero Waste Computing – SparseCCL for CPU

- SparseCCL runs **sequentially** over an image
  - We'll assume that SparseCCL is efficient
- How can we now run this efficiently on **multi-core** systems?
  - In other words: *where is the parallelism*?
- We have parallelism **between images (modules)**…
- …and **between events**!

SparseCCL: Connected Components Labeling and Analysis for sparse images

Arthur Hennequin[1,2], Ben Couturier[2], Vladimir V. Gligorov[3], Lionel Lacassagne[1]

[1]LIP6, Sorbonne Université, CNRS, Paris, France [2]CERN, Switzerland
[3]LPNHE, Sorbonne Université, Paris Diderot Sorbonne Paris Cité, CNRS/IN2P3, Paris, France
email: arthur.hennequin@lip6.fr, lionel.lacassagne@lip6.fr

***Abstract*—Connected components labeling and analysis for dense images have been extensively studied on a wide range of architectures. Some applications, like particles detectors in High Energy Physics, need to analyse many small and sparse images at high throughput. Because they process all pixels of the image, classic algorithms for dense images are inefficient on sparse data. We address this inefficiency by introducing a new algorithm specifically designed for sparse images. We show that we can further improve this sparse algorithm by specializing it for the data input format, avoiding a decoding step and processing multiple pixels at once. A benchmark on Intel and AMD CPUs shows that the algorithm is from ×1.6 to ×2.5 faster on sparse images.**

I. Introduction

In computer vision, Connected Component labeling (CCL) is a common and wide spread algorithm. Its purpose is to assign a unique label to each group of connected pixels. These groups of pixels, called Connected Components (CC), are then used for higher level tasks, like tracking, motion detection or optical character recognition. First instances of this algorithm were proposed by pioneers like Rosenfeld [20] or Haralick [8]. In High Energy Physics (HEP), CCL is used for tracking particles by labeling hits on the detectors' sensors to extract the real impact positions.

A CCL algorithm by itself, only provides the association of pixels, this is why it is followed by an analysis algorithm. The purpose of the analysis is to compute features of each CC, like the bounding box or the first statistical moments in order to compute the center of gravity. If naive algorithm perform the labeling first and then the analysis, the optimized algorithms do the analysis during the labeling. These algorithms are called *Connected Component Analysis* (CCA).

Most of CCL algorithms used to be sequential ones developed on single-core processors [9] [4]. Recently, new parallel algorithms were developed for multi-core processors [16] [7], SIMD processors [22] [14] [10], GPUs [18] [11] and FPGAs [13].

These algorithms are very efficient for natural images but not for very low density images (very few pixels set to one) like those generated in HEP experiments.

The case considered here is similar to matrix algebra. When a matrix has very few non-zero value, the classical dense structure and the classical dense algorithms turn out to be inefficient. The dense structure is replaced by various flavours of lists that hold the non-zero value and specialized or dedicated algorithms are designed to process these data efficiently. In the case of tracking hits on detectors' sensors, the same phenomenon happens: even if CCA algorithms are very fast, they are inefficient to cluster and label matrices of hits with a density around 0.5%.

In the Section II presents some classic connected components labeling algorithms. Section III introduces a new algorithm for connected components labeling of sparse images and its specialization for the pattern recognition of CERN's LHCb experiment. Finally, in Section IV, we evaluate this new algorithm and compare it to state-of-the-art.

II. Classic algorithms for dense images

In this section, we present three classes of connected components labeling algorithms.

A. One component at a time

In this first class of algorithm, we process one connected component at a time. The image is scanned one time and, for every foreground pixel encountered, a traversal of the connected component is done to label all the pixels. This algorithm and its variants are often called *flood fill* or sometimes *seed fill*. The traversal can be done using a stack in depth-first order or a queue, in breadth-first order. Implementations of algorithms of this class are found in [21] and [1]. This algorithm can be optimized by only adding, on top of the stack, the branching pixels – ie. the pixels that have more than one non-visited neighbour – and directly processing the others. Doing so, we avoid a store and a load for these pixels. If the image is sparse and if we have a list of pixel coordinates, we can directly start at known pixel positions avoiding the read of many background pixels. However, this does not prevent the test of every pixel on the contour of the connected component and it adds the cost of removing pixels from the list. An implementation of this algorithm specialized for the LHCb experiment is described in [3].

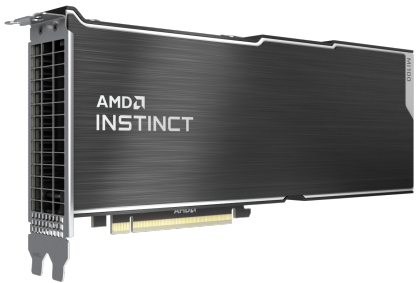
B. Iterative algorithms

The second class was introduced by Haralick [8]. Each pixel is initialized with a unique temporary label, then this label is propagated to the pixel's neighbors using local minimum

978-1-7281-4074-2/19/$31.00 ©2019 IEEE

Zero Waste Computing – SparseCCL for GPU

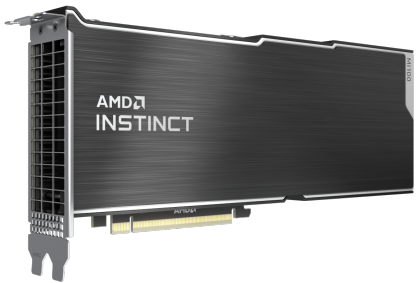
- Now, how can we do clustering on **graphics programming units (GPUs)**?
- Get a summer / technical / doctoral student to implement **SparseCCL** in **CUDA or HIP**!



Zero Waste Computing – SparseCCL for GPU

- Now, how can we do clustering on **graphics programming units (GPUs)**?
- Get a summer / technical / doctoral student to implement **SparseCCL** in **CUDA or HIP**!

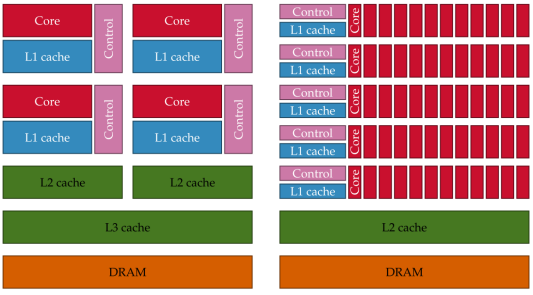
- It turns out this would be a **doomed effort**
- But we know this **before we write any code**
  - Analyse the properties of the software…
  - …as well as the hardware…
  - …and predict how they would interact!



Performance engineering gives us the tools to reduce waste of energy, compute time, *and* human resources

GPGPU Computing – Recap

- GPGPUs are designed to execute **similar tasks** in a **massively parallel** fashion
- This is achieved by sharing a **small amount of control** between a **large amount of compute**
- This makes individual “cores” less independent: **lock-step execution**!
- Beware NVIDIA marketing!



Zero Waste Computing – Hardware Differences

Differences in hardware require us to think differently about efficiency and zero-waste!

CPUs have…

- Complex pipelines & many ports
- Very large global memories
- Complex cache hierarchies; etc.

GPUs have…

- Large numbers of dependent “cores”
- Very-high-speed shared memories
- Coalesced load-store logic; etc.

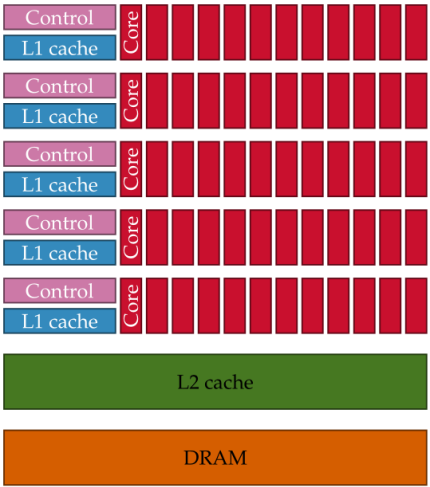
…so we must think (more) about…

- Instruction-level parallelism
- Coarse task-level parallelism
- Temporal data locality; etc.

- Thread imbalance & divergence
- Fine task-level parallelism
- Data access strides; etc.

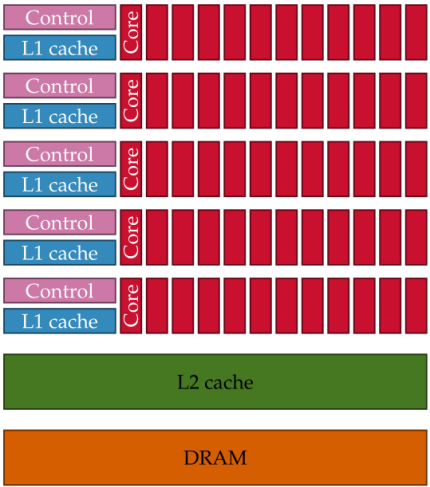
SparseCCL for GPU – Introduction

- How do we expose parallelism in **SparseCCL** for **GPUs**?
- Recall the design of GPUs: many **small cores that share control**



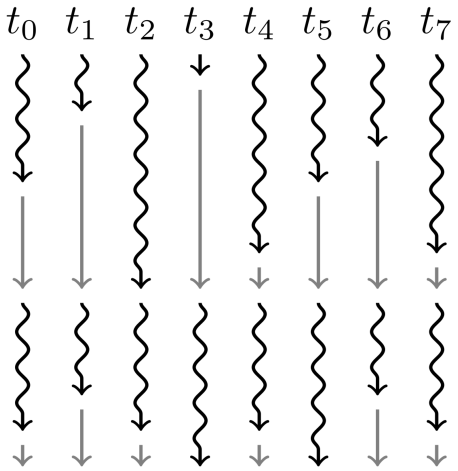
SparseCCL for GPU – Introduction

- How do we expose parallelism in **SparseCCL** for **GPUs**?
- Recall the design of GPUs: many **small cores that share control**
- Roughly two options: map each module onto one **thread**…
- …or onto one **thread group**



SparseCCL for GPU – Per-Thread

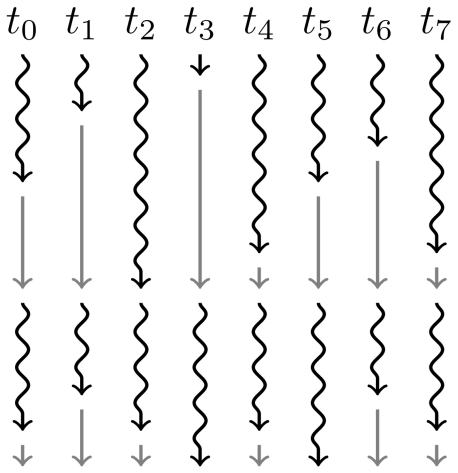
- Mapping **each module to one thread** feels natural
- …similar to how we parallelised for CPU!



SparseCCL for GPU – Per-Thread

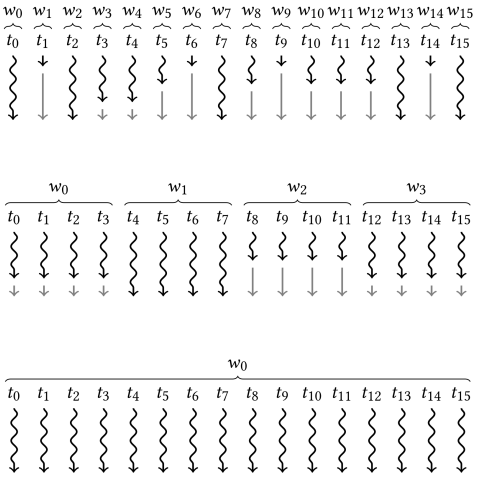
- Mapping **each module to one thread** feels natural
- ...similar to how we parallelised for CPU!

- But modules have **different hit counts**!
- This leads to **imbalance**: threads waiting (but still consuming power)
- Turns out we can also understand this behaviour through statistical **models**!



SparseCCL for GPU – Per-Group

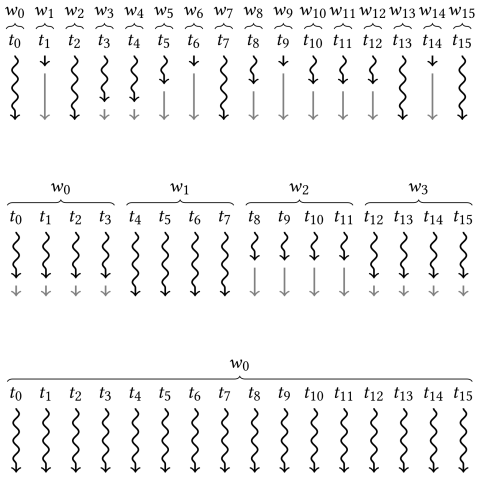
- We can also map one module onto a **group of threads**
- This is a powerful technique known as **thread refinement** (opp. **coarsening**)



SparseCCL for GPU – Per-Group

- We can also map one module onto a **group of threads**
- This is a powerful technique known as **thread refinement** (opp. **coarsening**)

- Does it actually help us here? Need to parallelise work over 32/64 threads
- Puts us in the **exact same problem as on CPU**: SparseCCL is sequential!

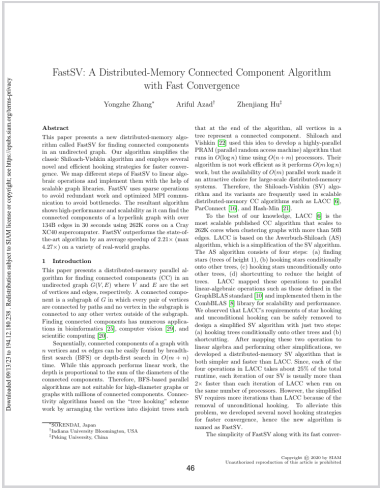


GPU Clustering – Where we are now

- To recap: we have **abstractly examined** two implementations:
  - Thread-per-module mapping suffers from **huge imbalance**
  - Group-per-module mapping suffers from **insufficient parallelism**
- Both implementations would be **wasteful**: **slow and power-inefficient**
- Implementing these kernels would take dozens of **person-hours**
- Performance engineering **from step one** saves us from wasting resources: *predictive power*

GPU Clustering – Finding a Way

- Performance engineering also allows us to **define requirements** which helps us to **find or design solutions**
- Prescriptive power!
- In this case, requirement for **shared memory**, **massive parallelism**, and **support for sparseness**



Downloaded 09/13/21 to 194.2.190.230. Redistribution subject to SIAM license or copyright; see http://epubs.siam.org/page-policy

FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence

Yongzhe Zhang* Ariful Azad† Zhenjiang Hu‡

Abstract

This paper presents a new distributed-memory algorithm called FastSV for finding connected components in an undirected graph. Our algorithm simplifies the classic Shiloach-Vishkin algorithm and employs several novel and efficient hooking strategies for faster convergence. We map different steps of FastSV to linear algebraic operations and implement them with the help of scalable graph libraries. FastSV uses sparse operations to avoid redundant work and optimized MPI communication to avoid bottlenecks. The resultant algorithm shows high performance and scalability as it can find the connected components of a hyperlink graph with over 124B edges in 30 seconds using 262K cores on a Cray XC40 supercomputer. FastSV outperforms the state-of-the-art algorithm by an average speedup of 2.21× (max 4.27×) on a variety of real-world graphs.

1 Introduction

This paper presents a distributed-memory parallel algorithm for finding connected components (CC) in an undirected graph $G(V, E)$ where V and E are the set of vertices and edges, respectively. A connected component is a subgraph of G in which every pair of vertices are connected by paths and no vertex in the subgraph is connected to any other vertex outside of the subgraph. Finding connected components has numerous applications in bioinformatics [23], computer vision [28], and scientific computing [29].

Sequentially, connected components of a graph with n vertices and m edges can be easily found by breadth-first search (BFS) or depth-first search in $O(m + n)$ time. While this approach performs linear work, the depth is proportional to the sum of the diameters of the connected components. Therefore, BFS-based parallel algorithms are not suitable for high-diameter graphs or graphs with millions of connected components. Connectivity algorithms based on the "tree hooking" scheme work by arranging the vertices into disjoint trees such that at the end of the algorithm, all vertices in a tree represent a connected component. Shiloach and Vishkin [22] used this idea to develop a highly-parallel PRAM (parallel random access machine) algorithm that runs in $O(\log n)$ time using $O(m + n)$ processors. Their algorithm is not work efficient as it performs $O(m \log n)$ work, but the scalability of $O(m)$ parallel work made it an attractive choice for large-scale distributed-memory systems. Therefore, the Shiloach-Vishkin (SV) algorithm and its variants are frequently used in scalable distributed-memory CC algorithms such as LACC [8], ParConnect [10], and Hash-Min [21].

To the best of our knowledge, LACC [8] is the most scalable published CC algorithm that scales to 262K cores when clustering graphs with more than 50B edges. LACC is based on the Awerbuch-Shiloach (AS) algorithm, which is a simplification of the SV algorithm. The AS algorithm consists of four steps: (a) finding stars (trees of height 1), (b) hooking stars conditionally onto other trees, (c) hooking stars unconditionally onto other trees, (d) shortcutting to reduce the height of trees. LACC mapped these operations to parallel linear-algebraic operations such as those defined in the GraphBLAS standard [16] and implemented them in the CombBLAS [8] library for scalability and performance. We observed that LACC's requirements of star hooking and unconditional hooking can be safely removed to design a simplified SV algorithm with just two steps: (a) hooking trees conditionally onto other trees and (b) shortcutting. After mapping these two operations to linear algebra and performing other simplifications, we developed a distributed-memory SV algorithm that is both simpler and faster than LACC. Since, each of the four operations in LACC takes about 25% of the total runtime, each iteration of our SV is usually more than $2\times$ faster than each iteration of LACC when run on the same number of processors. However, the simplified SV requires more iterations than LACC because of the removal of unconditional hooking. To alleviate this problem, we developed several novel hooking strategies for faster convergence, hence the new algorithm is named as FastSV.

The simplicity of FastSV along with its fast conver-

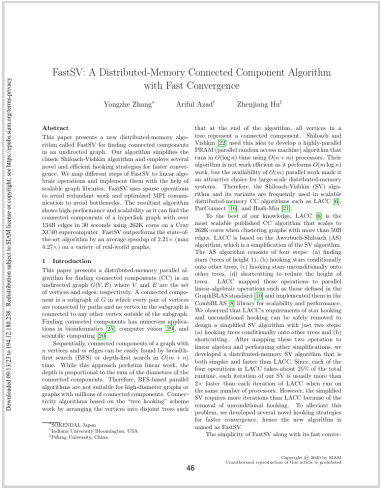
*RIKEN-DAI, Japan
†Indiana University Bloomington, USA
‡Peking University, China

Copyright © 2020 by SIAM
Unauthorized reproduction of this article is prohibited

46

GPU Clustering – FastSV

- We eventually settled on a **graph algorithm**: FastSV
- **Reduction** of sparse problem to a graph problem
  - Motivated by **cost modelling**
- Optimised for GPU execution using **shared memory**, **occupancy optimisation**, **thread coarsening**, **load balancing**, etc.



FastSV: A Distributed-Memory Connected Component Algorithm with Fast Convergence

Yongzhe Zhang* Ariful Azad† Zhenjiang Hu‡

Abstract

This paper presents a new distributed-memory algorithm called FastSV for finding connected components in an undirected graph. Our algorithm simplifies the classic Shiloach-Vishkin algorithm and employs several novel and efficient hooking strategies for faster convergence. We map different steps of FastSV to linear algebraic operations and implement them with the help of scalable graph libraries. FastSV uses sparse operations to avoid redundant work and optimized MPI communication to avoid bottlenecks. The resultant algorithm shows high performance and scalability as it can find the connected components of a hyperlink graph with over 124B edges in 30 seconds using 262K cores on a Cray XC40 supercomputer. FastSV outperforms the state-of-the-art algorithm by an average speedup of 2.21× (max 4.27×) on a variety of real-world graphs.

1 Introduction

This paper presents a distributed-memory parallel algorithm for finding connected components (CC) in an undirected graph $G(V, E)$ where V and E are the set of vertices and edges, respectively. A connected component is a subgraph of G in which every pair of vertices are connected by paths and no vertex in the subgraph is connected to any other vertex outside of the subgraph. Finding connected components has numerous applications in bioinformatics [23], computer vision [28], and scientific computing [29].

Sequentially, connected components of a graph with n vertices and m edges can be easily found by breadth-first search (BFS) or depth-first search in $O(m + n)$ time. While this approach performs linear work, the depth is proportional to the sum of the diameters of the connected components. Therefore, BFS-based parallel algorithms are not suitable for high-diameter graphs or graphs with millions of connected components. Connectivity algorithms based on the "tree hooking" scheme work by arranging the vertices into disjoint trees such that at the end of the algorithm, all vertices in a tree represent a connected component. Shiloach and Vishkin [22] used this idea to develop a highly-parallel PRAM (parallel random access machine) algorithm that runs in $O(\log n)$ time using $O(m + n)$ processors. Their algorithm is not work efficient as it performs $O(m \log n)$ work, but the scalability of $O(m)$ parallel work made it an attractive choice for large-scale distributed-memory systems. Therefore, the Shiloach-Vishkin (SV) algorithm and its variants are frequently used in scalable distributed-memory CC algorithms such as LACC [8], ParConnect [10], and Hash-Min [21].

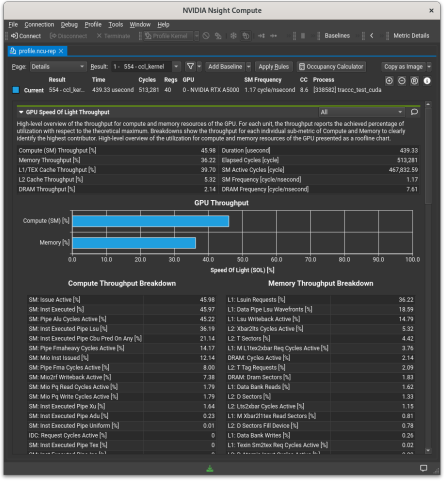
To the best of our knowledge, LACC [8] is the most scalable published CC algorithm that scales to 262K cores when clustering graphs with more than 50B edges. LACC is based on the Awerbuch-Shiloach (AS) algorithm, which is a simplification of the SV algorithm. The AS algorithm consists of four steps: (a) finding stars (trees of height 1), (b) hooking stars conditionally onto other trees, (c) hooking stars unconditionally onto other trees, (d) shortcutting to reduce the height of trees. LACC mapped these operations to parallel linear-algebraic operations such as those defined in the GraphBLAS standard [16] and implemented them in the CombBLAS [8] library for scalability and performance. We observed that LACC's requirements of star hooking and unconditional hooking can be safely removed to design a simplified SV algorithm with just two steps: (a) hooking trees conditionally onto other trees and (b) shortcutting. After mapping these two operations to linear algebra and performing other simplifications, we developed a distributed-memory SV algorithm that is both simpler and faster than LACC. Since, each of the four operations in LACC takes about 25% of the total runtime, each iteration of our SV is usually more than 2× faster than each iteration of LACC when run on the same number of processors. However, the simplified SV requires more iterations than LACC because of the removal of unconditional hooking. To alleviate this problem, we developed several novel hooking strategies for faster convergence, hence the new algorithm is named as FastSV.

The simplicity of FastSV along with its fast conver-

*RIKEN/AIL, Japan
†Indiana University Bloomington, USA
‡Peking University, China

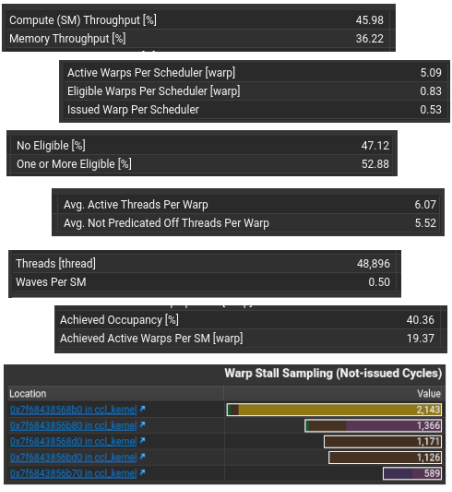
GPU Clustering – Evaluation

- Our solution perform up to **twice as well** as on an equivalent CPU
- **Descriptive tools** can now tell us what is still bottlenecking our code
  - In this case: NVIDIA Nsight Compute
- Approximately **50%** utilisation of resources (=50% waste!)



GPU Clustering – Future Work

- It remains to be seen if we will reach the **speed of light** for this kernel
- Important to posit requirements and goals in context (**Amdahl's law**)
- To improve, we **need** to apply **models**, **analyses**, and **techniques** from the PE domain!



Conclusions

- **Performance engineering** gives us a **toolbox** to **predict** performance, **prescribe** optimisation, and **describe behaviour**
- Many non-functional metrics:
  - **Running time**: more science in less time
  - **Power usage**: more science with reduced environmental impact
  - **Monetary cost**: more travel budget
  - **Implementation time**: more human resources to do science
- Requires us to **think carefully** and **from the beginning** about **software** and **hardware**