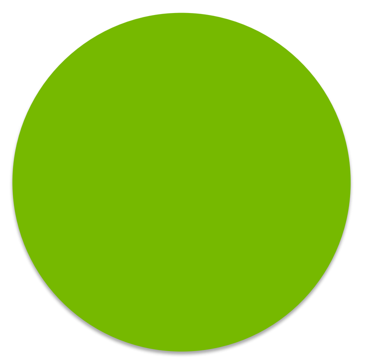
Computer Science
at the UNIVERSITY of VIRGINIA

High Performance and Scalable Radix Sorting for GPU Stream Architectures



Duane Merrill — dgm4d@virginia.edu

Andrew Grimshaw — grimshaw@virginia.edu

OVERVIEW

We have designed extremely efficient strategies for sorting large sequences of fixed-length keys (and values) using GPU stream processors. Our radix sorting methods exhibit speedup of up to 3.8x over the current state-of-the-art in GPU sorting. For this domain of sorting problems, we believe our sorting primitive to be the fastest available for any fully-programmable microarchitecture.

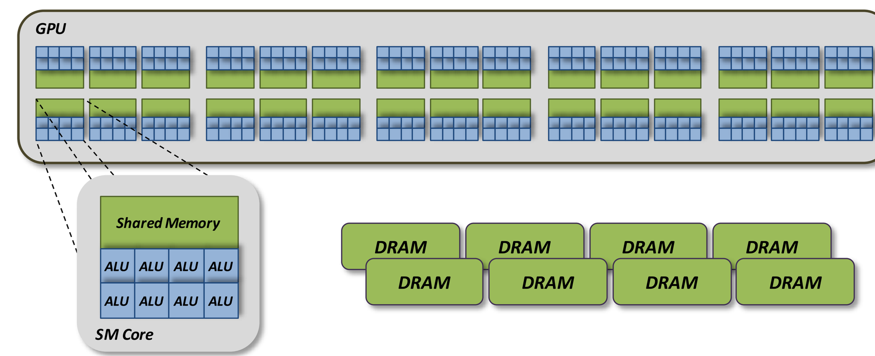
We refer to our approach as a *strategy* because we use a flexible hybrid composition of several different algorithms. The number of steps performed by each algorithmic phase can be configured to match the target platform, which allows us to construct a single implementation that scales well across all generations and configurations of programmable NVIDIA GPUs.

The need to rank and order data is pervasive, and sorting operations are fundamental to many algorithms. As an algorithmic primitive, GPU sorting facilitates many problems including:

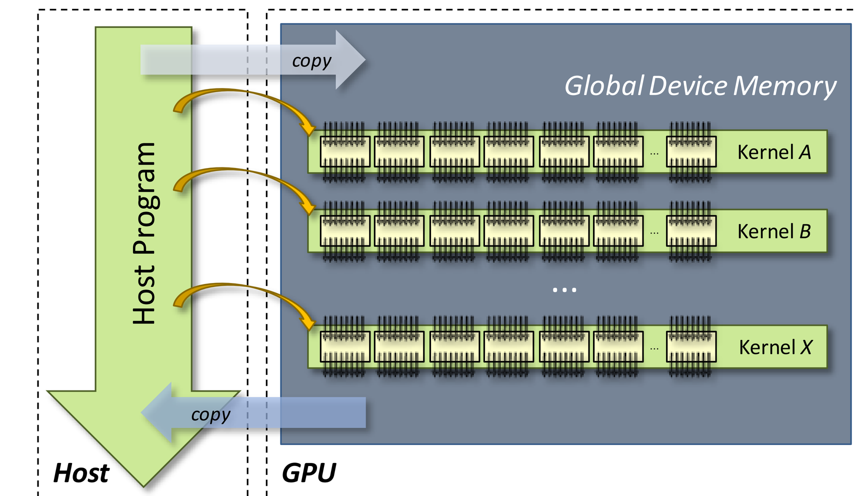
- Binary search
- Finding the closest pair
- Determining element uniqueness
- Finding the kth largest element
- Identifying outliers
- Shadow and transparency modeling
- Reyes rendering
- Volume rendering via ray-casting
- Particle rendering and animation
- Ray tracing
- Texture compression
- KD-tree construction
- Bounding volume hierarchy construction
- Collision detection
- Visibility culling
- Photon mapping
- Point cloud modeling
- Particle-based fluid simulation
- Parallel hashing
- Database acceleration
- Data mining
- Game engine AI

GPU STREAM PROCESSORS

The GPU is capable of efficiently executing large quantities of concurrent, ultra-fine-grained tasks. They are SPMD (single program, multiple data) machines having many hardware-scheduled execution contexts, or *threads*, that all run copies of the same imperative program, or *kernel*. The host orchestrates a *stream* of global data flow by repeatedly invoking new kernel instances, each of which is initially presented with a consistent view of the results from the previous.



GPU processor organization entails a collection of multithreaded cores, each of which is comprised of a set of homogeneous processing elements. These SM cores employ local SIMD (single instruction, multiple data) techniques in which a single instruction stream is executed by a fixed-size grouping of threads called a *warp*. Each SM core maintains and schedules amongst the execution contexts of many warps. This translates into tens of warp contexts per core, and tens-of-thousands of thread contexts per GPU processor for very high computational bandwidths.



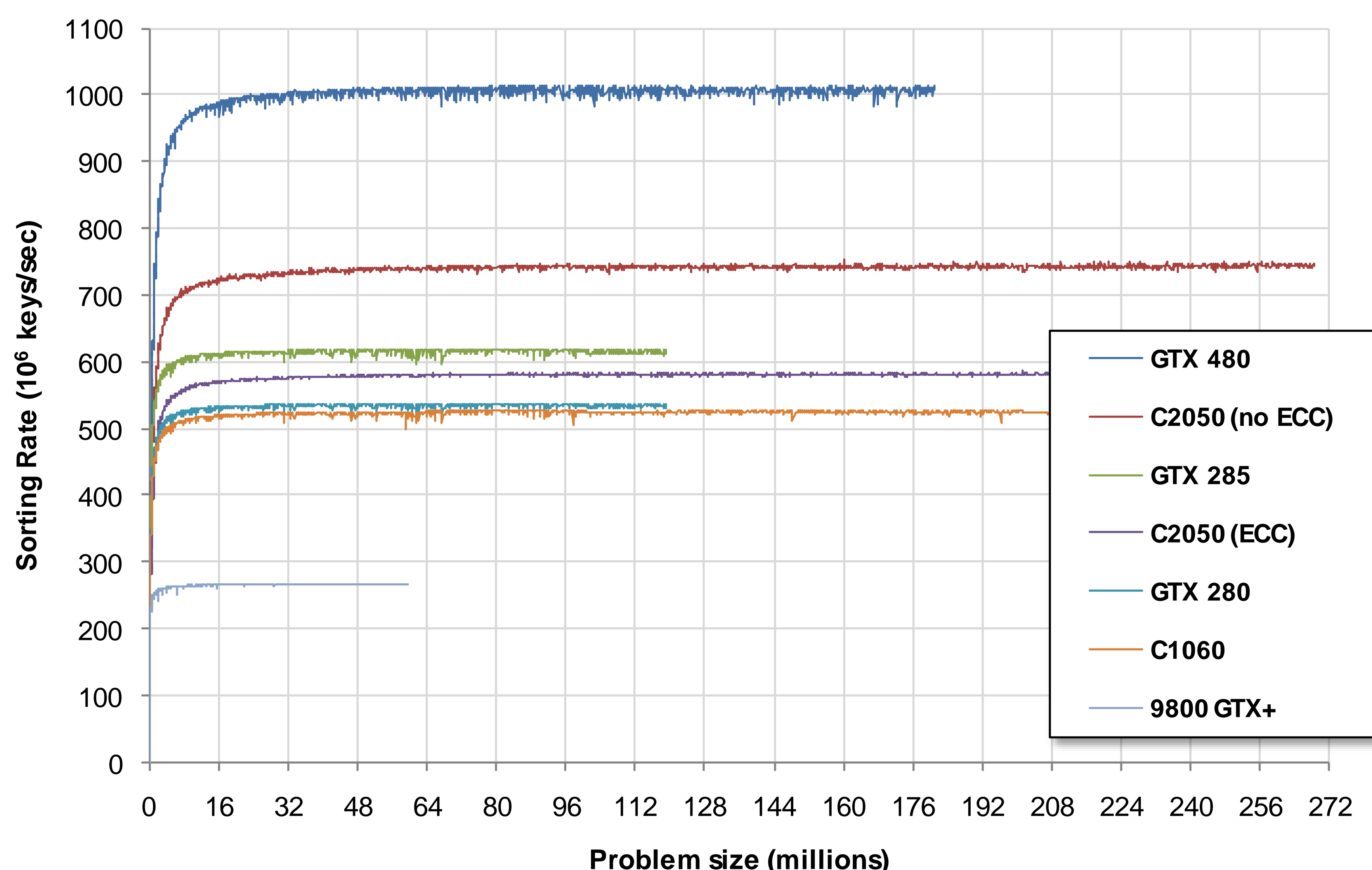
OUR SRTS RADIX SORTING PERFORMANCE

Our implementation (SRTS) demonstrates average sorting rates of up to 1,005 million 32-bit keys per second, and 775 million 32-bit key-value pairs per second. We provide multiple factors of speedup over the state-of-the-art GPU sorting routines provided by the CUDPP data parallel primitives library.

We also revisit sorting comparisons in the literature between many-core CPU and GPU architectures. Our speedups show:

- G80-based GPUs to outperform Intel Core2 quad-core CPUs
- GT200-based GPUs to outperform Core i7 quad-core CPUs
- GF100-based GPUs to outperform Intel 32-core Knight's Ferry (Larrabee derivative) by as much as 1.8x.

Device	Key-value Rate (10^6 pairs / sec)			Keys-only Rate (10^6 keys / sec)		
Name	CUDPP Radix	SRTS Radix	(speedup)	CUDPP Radix	SRTS Radix	(speedup)
NVIDIA GTX 480		775			1005	
NVIDIA Tesla C2050		581			742	
NVIDIA GTX 285	134	490	(3.7x)	199	615	(2.8x)
NVIDIA GTX 280	117	449	(3.8x)	184	534	(2.6x)
NVIDIA Tesla C1060	111	333	(3.0x)	176	524	(2.7x)
NVIDIA 9800 GTX+	82	189	(2.0x)	111	265	(2.0x)
NVIDIA 8800 GT	63	129	(2.1x)	83	171	(2.1x)
NVIDIA Quadro FX5600	55	110	(2.0x)	66	147	(2.2x)
Intel 32-core Knight's Ferry MIC					560	
Intel quad-core i7					240	
Intel Q9550 quad-core					138	

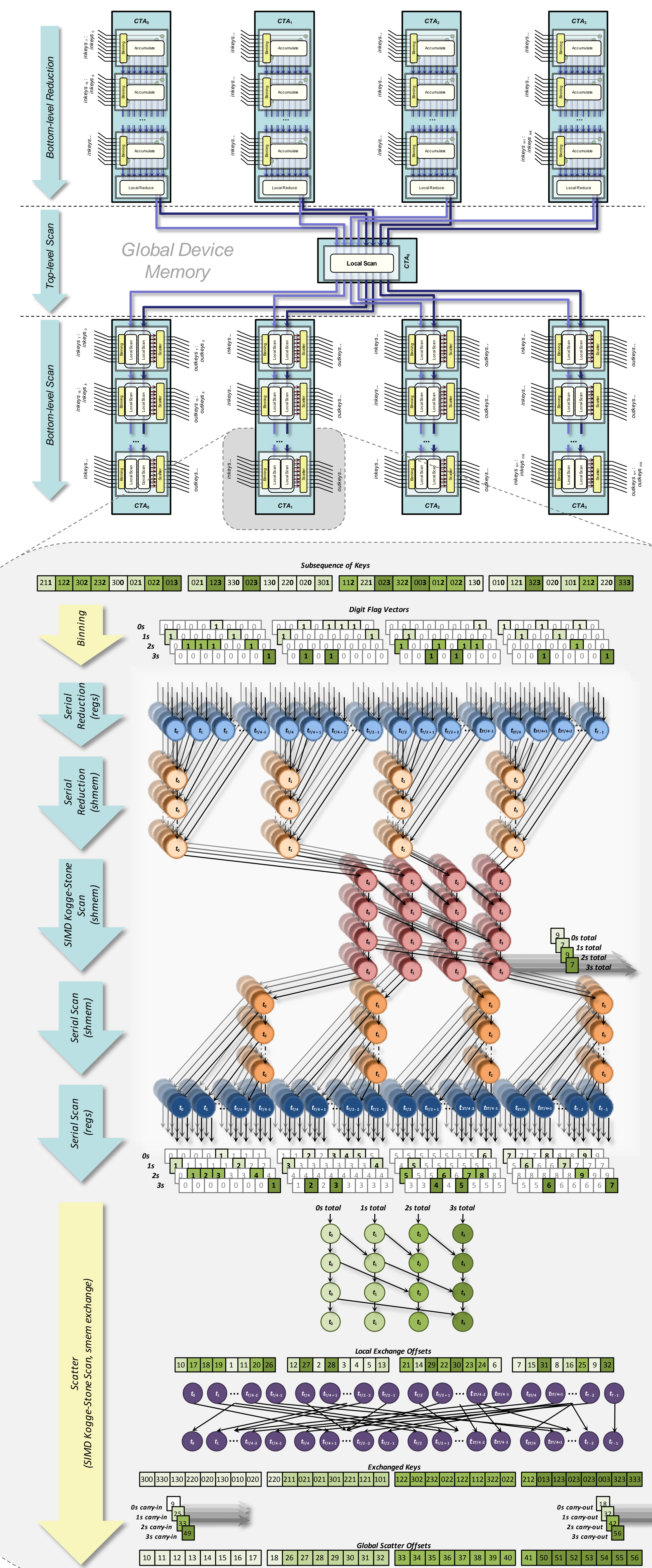


RADIX SORTING METHOD

The radix sorting method works by iterating over the digit-places of the keys from least-significant to most-significant. For each digit-place, the method performs a stable *distribution sort* of the keys based upon their digit at that digit-place. Given an *n*-element sequence of *k*-bit keys and a radix $r = 2^d$, a radix sort of these keys will require k/d iterations of a distribution sort over all *n* keys.

The distribution sort is the fundamental component of the radix sorting method. In a data-parallel, shared-memory decomposition, each logical processor gathers its key, decodes the specific digit at the given digit-place, and then must cooperate with other processors to determine where the key should be relocated. The relocation offset will be the key's global rank, i.e., the number of keys with "lower" digits at that digit place plus the number of keys having the same digit, yet occurring earlier in the sequence.

DATAFLOW REPRESENTATION OF THE DISTRIBUTION SORT



CRITICAL INSIGHTS

Our performance is derived from our ability to efficiently determine relocation offsets for scattering keys. To perform these distribution sorting passes, we have constructed a parallel prefix scan primitive that has been augmented in two ways:

1. ***Kernel fusion (left)***. We embed logic for generating and consuming prefix scan problems and results within the scan kernel itself.
2. ***Increased granularity (right)***. We perform multiple related, concurrent prefix scans in order increase the number of radix numerals and thus decrease the number of digit places we must iterate over.