Modular Sorting on a Fine-Grained Many-Core Processor Array

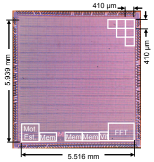
UCDAVIS COLLEGE of ENGINEERING

Aaron Stillmaker and Bevan Baas

VLSI Computation Laboratory
Department of Electrical and Computer Engineering
University of California, Davis

Fine-Grained Many-Core Processor Arrays

- **Very efficient programmable processors**
  - Very small area, simple architecture
  - Very low power when active (individual clock frequency and supply voltage scaling)
  - Very low power when idle (individual clock oscillator halting)
- **No specialized instructions**
  - Special-purpose accelerators if warranted
- **On-chip shared memories**
- **Shown to work very well for DSP, multimedia, embedded applications**
- **This project examines domain extension to enterprise workloads**
  - Many-core array as a co-processor
  - Many-core array as a functional unit
  - Many-core array computes entire applications or critical computational kernels
- **Project commenced June 2010**

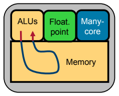


AsAP2 Single Tile (164 total)	
Area	0.17 mm²
Transistors	325,000
CMOS Tech.	65 nm low-leakage
Max. frequency	1.19 GHz @ 1.3 V
Power (100% active)	47 mW @ 1.06 GHz, 1.2 V 3.4 mW @ 260 MHz, 0.75 V 608 µW @ 66 MHz, 0.675 V

Many-Core Array as a Co-Processor

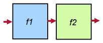
- **General-purpose CPU example kernel sort code**

```
if a < b
    out1 = a;
    out2 = b;
else
    out1 = b;
    out2 = a;
```



  - Datapath ↔ memory loop is “long” and built with high power circuits
- **Example code with many-core co-processor or functional unit**

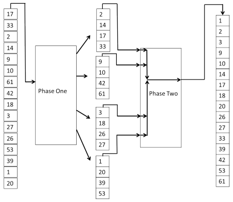
```
write 100kB unsorted data;
read  100kB sorted data;
```



  - Fine-grain cores typically abut and are built with very efficient circuits
- **Key points**
  - Minimize total data movement distance in computational kernels
  - Can view as a highly-flexible programmable functional unit
  - Challenge: extracting kernels!

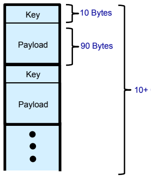
Sorts With Very Large Data Sets

- **“External” sorts typically utilize two distinct phases:**
  1) Sort all records that fit in memory
     - We initially focus on this phase since it is where the greatest improvements appears to be possible
  2) Successively merge lists into one final sorted list



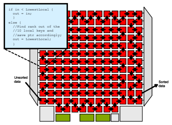
Phase 1 Sorting on AsAP2

- **“Database” records**
  - 10 GB – 1 TB of data
  - 100 Byte records
- **This work focuses on the first phase of external sorting**
  - Large part of the sort, so we targeted this to give the greatest returns
  - Serial CPUs already handle merging large lists well
- **Streaming sorting**
  - Keys are kept with payloads so that unsorted data can be input, and sorted data is streamed out
  - No need for a CPU to reattach payloads



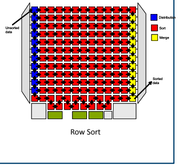
Sorting on AsAP2

- **“Snake Sort” algorithm**
  - Insertion sort inside each processor functions like a bubble sort on the chip
  - Exact same code in every processor
  - Simple but not highly efficient
    - Every record passes through every core
    - Every record is compared by every core
- **“Row Sort” algorithm**
  - Round-robin *distribution* processors
  - Linear *sort* processors in rows
  - *Merging* processors
  - “Flush” command packet signals sorting processors that the block has completed and is ready for merging
- **No globally-accessible memory such as a cache**
  - Extremely power efficient
  - Limited sorting algorithms possible



```
if in < LowestLocal {
  out = in;
  }
else {
  //Find rank out of the
  //10 local keys and
  //save ptr accordingly;
  out = LowestLocal;
  }
```

Snake Sort



Row Sort

SAISort Kernel

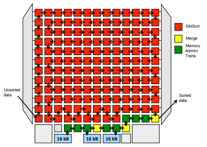
- **Pseudo Code of the main Serial Array of Insertion Sort (SAISort) kernel used in all of the sorts:**

Algorithm 1 SAISort

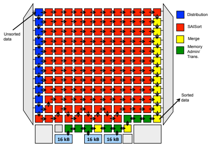
```
while true do
  if $inputTag \neq$ Reset then
    if $input \leq lowest$ then
      $output \leftarrow input$
    else
      $output \leftarrow lowest$
      Place $input$ in appropriate position
    end if
  else
    Save the number of records to pass
    Output Reset command
    Add number of records in processor to the number or records to pass, and output
    Flush all records in processor
    Pass given number of records straight from the input to output
  end if
end while
```

SAISort Phase 1 Sorting on AsAP2: Using Large Shared Memories

- **“Snake Sort” and “Row Sort” using three 16 kB shared memories at the bottom of the array**
  - Runs are saved in memories, then merged after the memories are filled
  - Processors are required for transmission and merging



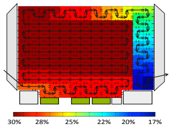
Snake Sort



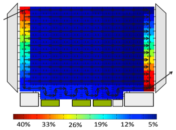
Row Sort

Processor Activity During Sort

- **Heatmaps showing the active percentages of the processors**
  - In the “Snake Sort”, activity reaches a plateau of 30% for most of the mid processors
  - In the “Row Sort”, the input and output processors are the bottleneck
  - The processors stall when waiting for input or output
    - Each processor consumes essentially zero power while stalled
  - This shows one run, with multiple runs, there would be overlap, removing some of the underutilized processors



Snake Sort



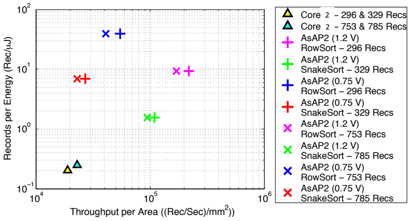
Row Sort

Sorting Results With 100-Byte Records

Records Per Block	Processor		Thruput (1,000 rec/sec)	Thruput per Area ((rec/sec)/mm^2)	Energy per Rec (nJ/rec)
296	Intel Core 2 Duo quicksort		3,700	23,000	4,000
	AsAP2, Row Sort	1.2 V	**8,900**	**220,000**	90.8
		0.75 V	2,200	55,000	**19.8**
329	Intel Core 2 Duo quicksort		3,700	23,000	4,000
	AsAP2, Snake Sort	1.2 V	4,400	110,000	670
		0.75 V	1,070	27,000	146
753	Intel Core 2 Duo quicksort		3,000	19,000	4,900
	AsAP2, Row Sort w/ On Chip Memories	1.2 V	6,600	170,000	108
		0.75 V	1,600	41,000	25.6
785	Intel Core 2 Duo quicksort		3,000	19,000	4,900
	AsAP2, Snake Sort w/ On Chip Memories	1.2 V	3,800	95,000	646
		0.75 V	910	23,000	143

- **Sorting 10 GB of 100-Byte records including a 10-Byte key**
  - All Intel Core 2 Duo results utilize an unoptimized quicksort
    - Results are scaled for technology
  - AsAP2 Sorting uses up to 200x less energy
  - AsAP2 Sorting recognizes up to 9x higher throughput per area

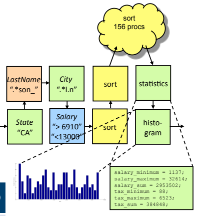
Sorting Results With 100-Byte Records



Example Enterprise Application: Database RegExp + Sort + Statistics

- **200-Byte records**
- **Search query:**
  - `“CA” == State && “.*son_” ϵ LastName && “.*l.n”  ϵ City && 6910 < Salary < 13000`
- **Mapping algorithm**
  - Each search portion is mapped to a core
  - Inter-core data: complete record plus a pointer to the next search’s beginning character → highly reusable programs

Field	Size (Bytes)
Employee ID	20
FirstName	20
LastName	20
State	4
City	36
Salary	2
Tax with.	2
allow.	4
Birth month	4
Birth day	8
Birth year	20
Phone, home	20
Phone, cell	20
Phone, work	20
Phone, fax	

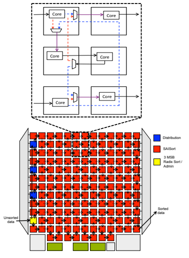


```
salary_minimum = 1137;
salary_maximum = 32614;
salary_sum = 2953502;
tax_minimum = 88;
tax_maximum = 6523;
tax_sum = 384848;
```

Supply Voltage	Max Freq.	Thruput (MB/s)	Power (mW)	Energy (nJ/Byte)
1.2 V	1070 MHz	1520	44	27
0.75 V	260 MHz	369	3.1	8.1
0.675 V	66 MHz	143	0.56	5.7

Current / Future Work

- **“Dynamic Sort” algorithm**
  - Radix sort performed on 3 MSBs of incoming keys
  - Admin processor will dynamically allocate the processors in between rows to the row filling up the fastest
    - Goal is to have rows that can dynamically resize based on the demand
    - Should show speedup for evenly distributed records, as it is similar to the “Row Sort” without the need for merging in the end
- **Core Scaling**
  - Run simulations on how the sorting versions will perform with different sized processor arrays



Dynamic Sort

VLSI Computation LAB