



FernUniversität
in Hagen

Bitslice Base64 Encoding on GPU

M. Rosenbaum, C. Heßeling, *J. Keller*



PARS-Workshop
Sept 18, 2026

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Prof. Dr. Jörg Keller



Structure

- **Motivation**
- **Background**
- **Approach**
- **Experiments**
- **Conclusions and Outlook**

Motivation

- Hierarchical structure of computing systems:
network of nodes
nodes with multicore CPU and accelerator (GPU)
each core/thread with normal and vector registers
- Which workload placed onto which part?
- Rule of thumb: regular, data-parallel computations offloaded to GPU
 - ➔ done by Jin and Finkel (2019) for Base64 encoding
 - ➔ yet: does not really use 32-bit architecture of GPU



Motivation

- Idea from encryption (Biham 1997):
many identical computations not well suited for software implementation
→ do bitslice implementation
- Nice side-effects: avoids control-flow divergence
avoids some side channels
- Approach (easy for table-based computation):
express computation as bundle of boolean functions $f_i: \{0,1\}^6 \rightarrow \{0,1\}$, $i=0\dots7$
start from function tables
synthesize formulae for all f_i
do n computations at once on n -bit registers! (Yet conversion overhead before + after)



Motivation

- Idea from encryption (Biham 1997):
many identical computations not well suited for software implementation
→ do bitslice implementation
- Nice side-effects: avoids control-flow divergence
avoids some side channels
- Approach (easy for table-based computation):
express computation as bundle of boolean functions $f_i: \{0,1\}^6 \rightarrow \{0,1\}$, $i=0\dots7$
start from function tables
synthesize formulae for all f_i
do n computations at once on n -bit registers! (Yet conversion overhead before + after)



Useful on
GPU?
(Nishikawa 2017)

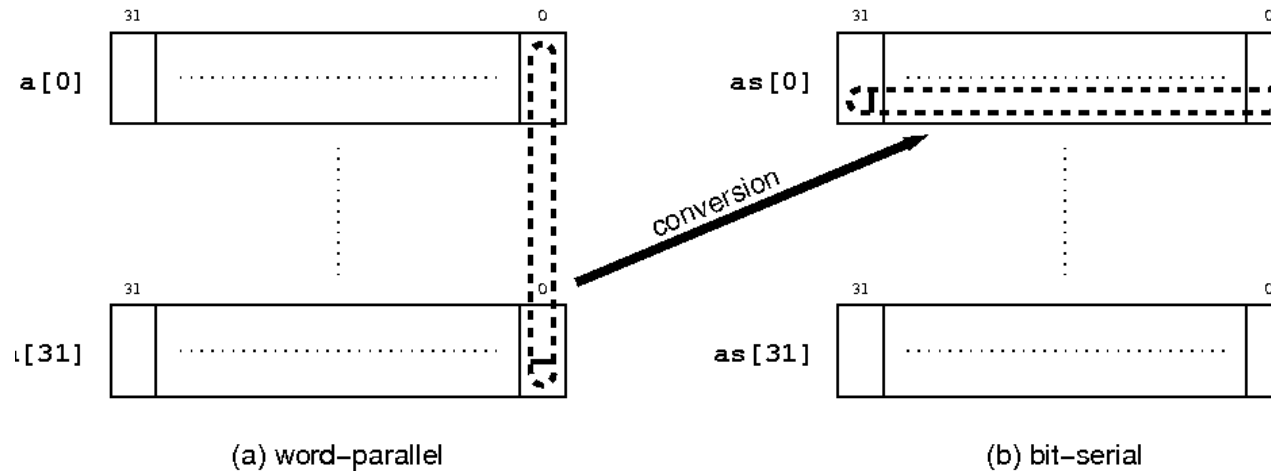
Base64 Encoding

- Transport binary data via text-based media
- Basic idea: take 6 bits of input
convert into printable ASCII character (8 bit)
normally: via table lookup
- Input unit: $4 \cdot 6 = 24$ bit = 3 bytes
Output unit: $4 \cdot 8 = 32$ bit = 4 bytes
- Awkward: 1st block=bits 7...2 of byte 1
2nd block= bits 1...0 of byte 1, bits 7...4 of byte 2
3rd block= bits 3...0 of byte 2, bits 7...6 of byte 3
4th block= bit 5...0 of byte 3

6-bit index		Assigned Character	ASCII code	
hex	$x_5 \cdots x_0$		hex	$y_7 \cdots y_0$
0x00	000000	A	0x41	0100 0001
⋮	⋮	⋮	⋮	⋮
0x19	011001	Z	0x5a	0101 1010
0x1a	011010	a	0x61	0110 0001
⋮	⋮	⋮	⋮	⋮
0x33	110011	z	0x7a	0111 1010
0x34	110100	0	0x30	0011 0000
⋮	⋮	⋮	⋮	⋮
0x3d	111101	9	0x39	0011 1001
0x3e	111110	+	0x2b	0010 1011
0x3f	111111	/	0x2f	0010 1111

Overhead

- Input data must be transformed into required format
 - ➔ bits 31 to 0 of $a[i]$ must go to bit i of $as[31]$ to $as[0]$
 - ➔ transposition
- Same for output data (transposition is self-inverse)



Bitmatrix Transposition

- Bitmatrix: each matrix row in one variable
→ row exchange simple, column exchange not
- Recursion if matrix size is power of 2: Sub-matrices are $n/2 \times n/2$

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix}$$

- $A^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \end{pmatrix}$

→ Transpose all sub-matrices, then exchange non-diagonal sub-matrices

Bitmatrix Transposition

- End of recursion: 1x1 sub-matrix ist identical under transposition
- Unroll recursion: Sequence of exchanges, 1bit blocks, 2bit blocks, 4bit blocks,...
a0a1a2a3a4a5 a0b0a2b2a4b4
b0b1b2b3b4b5 → a1b1a3b3a5b5
- SWAPMOVE(a,b):
tmp = (a & 0x5...5) | ((b & 0x5...5) << 1)
b = ((a & ~0x5...5) >> 1) | (b & ~0x5...5)
a = tmp
General: SWAPMOVEr does shifts over 2^r and has masks with alternating 2^r 1s + 0s

Bitmatrix Transposition

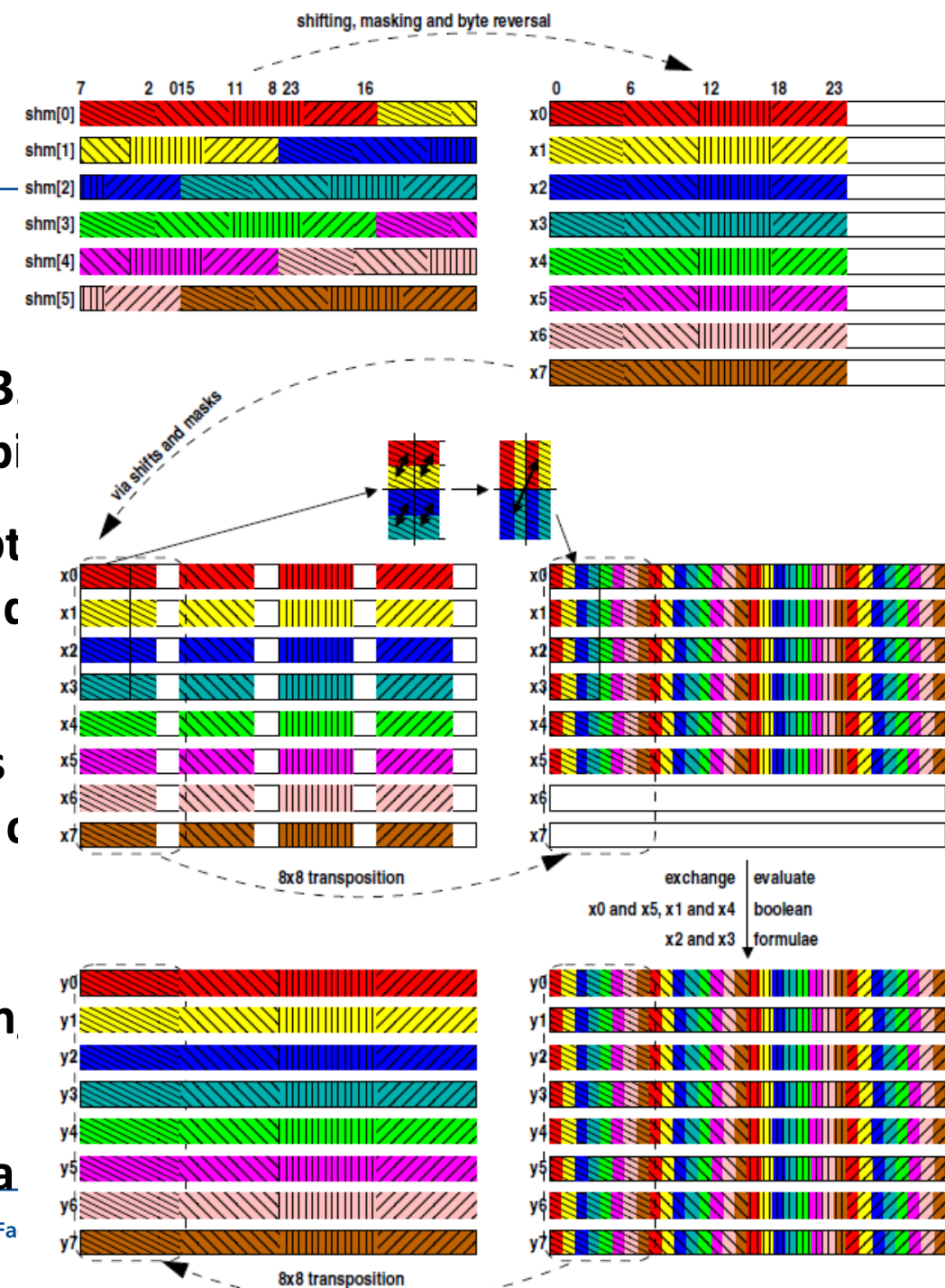
- for $r=0$ to $\log_2(n)-1$ do
 $k = 2^r$
 // swap non-diagonal sub-matrices of size $k \times k$
 for $l=0$ to $n-1$ step $n/2^{r+1}$ do // start swaps at row l
 for $m=0$ to 2^r-1 do // start swapmove at rows $l+m$ and $l+m+k$
 swapmove(&row[$l+m$], &row[$l+m+k$], mask[r], r)
- Complexity $O(n \log n)$ for $n \times n$ bitmatrix

Approach

- Each GPU thread uses 32-bit registers → does 32 encodings at once
input size $32 \times 6 \text{ bit} = 24 \text{ byte}$, output size $32 \times 8 \text{ bit} = 32 \text{ byte}$
- Avoid loads from GPU global mem with non-optimal stride and byte transfers
→ thread block loads 6 unsigned ints per thread into GPU shared memory
- Each thread
re-arranges 6 unsigned ints into 8 unsigned ints (each with 24 bits used)
mirrors bits in each byte to have 6 bit blocks in consecutive positions
inserts 2-bit gaps
does $8 \times 8(x4)$ bitmatrix transpose
evaluates boolean formulae for base64 encoding (61 op.s, with var.s i and 6-i exchanged)
does $8 \times 8(x4)$ bitmatrix transpose
- Results are written back to GPU global mem via shared mem

Approach

- Each GPU thread uses 32-bit registers → does 3 input size 32×6 bit = 24 byte, output size 32×8 bit
- Avoid loads from GPU global mem with non-opt → thread block loads 6 unsigned ints per thread
- Each thread re-arranges 6 unsigned ints into 8 unsigned ints mirrors bits in each byte to have 6 bit blocks in c inserts 2-bit gaps does $8 \times 8(x4)$ bitmatrix transpose evaluates boolean formulae for base64 encoding does $8 \times 8(x4)$ bitmatrix transpose
- Results are written back to GPU global mem via



Experiments on GeForce RTX 5070 Ti

- Compare bitslice to table-based implementation
- Test influence of block size and input size

Input	Size of CUDA Block	Bitslice		Jin and Finkel	
		Avg Runtime (ms)	Std.dev.	Avg. Runtime (ms)	Std.dev.
$2^{22} \cdot 3$	32	0.0956	0.0052	0.2638	0.0106
$2^{22} \cdot 3$	64	0.0987	0.0068	0.2673	0.0148
$2^{22} \cdot 3$	128	0.0994	0.0092	0.2691	0.0114
$2^{20} \cdot 3$	32	0.0799	0.0056	0.1552	0.0195
$2^{20} \cdot 3$	64	0.0792	0.0047	0.1538	0.0184
$2^{20} \cdot 3$	128	0.0796	0.0053	0.1598	0.0206
$2^{18} \cdot 3$	32	0.0720	0.0034	0.1072	0.0148
$2^{18} \cdot 3$	64	0.0733	0.0035	0.1086	0.0134
$2^{18} \cdot 3$	128	0.0713	0.0030	0.1098	0.0170
$2^{16} \cdot 3$	32	0.0695	0.0034	0.0971	0.0091
$2^{16} \cdot 3$	64	0.0690	0.0035	0.0955	0.0109
$2^{16} \cdot 3$	128	0.0683	0.0032	0.0958	0.0043

Experiments on GeForce RTX 5070 Ti

- Compare bitslice to table-based implementation
- Test influence of block size and input size

Little influence of
block size

Input	Size of CUDA Block	Bitslice		Jin and Finkel	
		Avg Runtime (ms)	Std.dev.	Avg. Runtime (ms)	Std.dev.
$2^{22} \cdot 3$	32	0.0956	0.0052	0.2638	0.0106
$2^{22} \cdot 3$	64	0.0987	0.0068	0.2673	0.0148
$2^{22} \cdot 3$	128	0.0994	0.0092	0.2691	0.0114
$2^{20} \cdot 3$	32	0.0799	0.0056	0.1552	0.0195
$2^{20} \cdot 3$	64	0.0792	0.0047	0.1538	0.0184
$2^{20} \cdot 3$	128	0.0796	0.0053	0.1598	0.0206
$2^{18} \cdot 3$	32	0.0720	0.0034	0.1072	0.0148
$2^{18} \cdot 3$	64	0.0733	0.0035	0.1086	0.0134
$2^{18} \cdot 3$	128	0.0713	0.0030	0.1098	0.0170
$2^{16} \cdot 3$	32	0.0695	0.0034	0.0971	0.0091
$2^{16} \cdot 3$	64	0.0690	0.0035	0.0955	0.0109
$2^{16} \cdot 3$	128	0.0683	0.0032	0.0958	0.0043

Experiments on GeForce RTX 5070 Ti

- Compare bitslice to table-based implementation
- Test influence of block size and input size

Little influence of
block size

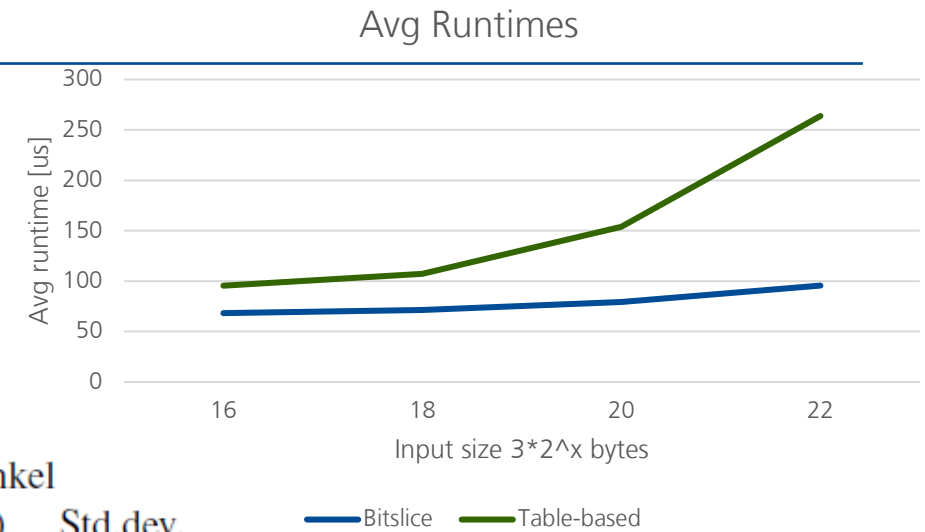
Input	Size of CUDA Block	Bitslice		Jin and Finkel	
		Avg Runtime (ms)	Std.dev.	Avg. Runtime (ms)	Std.dev.
$2^{22} \cdot 3$	32	0.0956	0.0052	0.2638	0.0106
$2^{22} \cdot 3$	64	0.0987	0.0068	0.2673	0.0148
$2^{22} \cdot 3$	128	0.0994	0.0092	0.2691	0.0114
$2^{20} \cdot 3$	32	0.0799	0.0056	0.1552	0.0195
$2^{20} \cdot 3$	64	0.0792	0.0047	0.1538	0.0184
$2^{20} \cdot 3$	128	0.0796	0.0053	0.1598	0.0206
$2^{18} \cdot 3$	32	0.0720	0.0034	0.1072	0.0148
$2^{18} \cdot 3$	64	0.0733	0.0035	0.1086	0.0134
$2^{18} \cdot 3$	128	0.0713	0.0030	0.1098	0.0170
$2^{16} \cdot 3$	32	0.0695	0.0034	0.0971	0.0091
$2^{16} \cdot 3$	64	0.0690	0.0035	0.0955	0.0109
$2^{16} \cdot 3$	128	0.0683	0.0032	0.0958	0.0043

Influence of
input size varies

Experiments on GeForce RTX 5070 Ti

- Compare bitslice to table-based implementation
- Test influence of block size and input size

Input	Size of CUDA Block	Bitslice		Jin and Finkel	
		Avg Runtime (ms)	Std.dev.	Avg. Runtime (ms)	Std.dev.
$2^{22} \cdot 3$	32	0.0956	0.0052	0.2638	0.0106
$2^{22} \cdot 3$	64	0.0987	0.0068	0.2673	0.0148
$2^{22} \cdot 3$	128	0.0994	0.0092	0.2691	0.0114
$2^{20} \cdot 3$	32	0.0799	0.0056	0.1552	0.0195
$2^{20} \cdot 3$	64	0.0792	0.0047	0.1538	0.0184
$2^{20} \cdot 3$	128	0.0796	0.0053	0.1598	0.0206
$2^{18} \cdot 3$	32	0.0720	0.0034	0.1072	0.0148
$2^{18} \cdot 3$	64	0.0733	0.0035	0.1086	0.0134
$2^{18} \cdot 3$	128	0.0713	0.0030	0.1098	0.0170
$2^{16} \cdot 3$	32	0.0695	0.0034	0.0971	0.0091
$2^{16} \cdot 3$	64	0.0690	0.0035	0.0955	0.0109
$2^{16} \cdot 3$	128	0.0683	0.0032	0.0958	0.0043



Conclusions and Outlook

- Bitslice implementations can be useful beyond encryption
➔ another thread-level parallelism next to vector registers? Automatism to convert?
- Can vector registers on CPUs be used as well for bitslice implementations?
- Which operations are needed for the conversions? (shifts over word borders)
- Should base64 be re-defined to improve performance?
Bit positions, ASCII code mapping

Thanks for your kind attention!

Reach me via

joerg.keller@fernuni-hagen.de