Base64 Encoding on Heterogeneous Computing Platforms

Zheming Jin

Acknowledgement: This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

Motivation

- Base64 format has many applications
  - Embedding resources within HTML page
  - Web storage stores Base64-encoded data in the web browsers
  - Base64 strings for binary data in database systems
- Heterogeneous computing for Base64 Encoding
  - Previous studies focused on vectorizations on CPUs
  - Improve the streaming application with concurrency
  - Explore what degree of concurrency CUDA and OpenCL streams can achieve
- Performance and power tradeoffs
  - We expect the GPU is faster than the FPGA in raw performance
  - The FPGA has an edge is power saving

Contributions

- Describe the transformations
  - From the algorithm to CUDA and OpenCL kernels for heterogeneous computing devices
- Optimize the OpenCL application
  - CUDA/OpenCL streams
  - Loop transformations
  - Kernel optimizations
- Evaluate the impact of the optimizations upon performance
  - Performance comparison on the CPU, GPU, and FPGA

Background (Base64 Encoding)

Algorithm 1: Base64 encoding
Input: A stream *s* of *n* bytes, indexed as $s_0, s_1, \ldots, s_{n-1}$
Output: A stream *o* of *m* bytes, indexed as $o_0, o_1, \ldots, o_{m-1}$

```
for ($i$ = 0; $i$ < $n$; $i$ = $i$ + 3) do
 $o_i$   = $F(s_i \div 4)$
 $o_{i+1}$ = $F(((s_i \times 16) \bmod 64) + (s_{i+1} \div 16))$
 $o_{i+2}$ = $F((s_{i+1} \times 4) \bmod 64) + (s_{i+2} \div 64))$
 $o_{i+3}$ = $F(s_{i+2} \bmod 64)$
end for
```

each block of three input bytes (s_i, s_{i+1}, s_{i+2}) is combined arithmetically to four 6-bit words (o_i, o_{i+1}, o_{i+2}, o_{i+3}).

```
$pad$ = $n \bmod 3$
if $pad$ != 0 then
  $o_i$ = $F(s_i \div 4)$
  if $pad$ = 1 then
    $o_{i+1}$ = $F(((s_i \times 16) \bmod 64)$
    $o_{i+2}$ = '='
   else if $pad$ = 2 then
     $o_{i+1}$ = $F(((s_i \times 16) \bmod 64) + (s_{i+1} \div 16))$
     $o_{i+2}$ = $F(((s_{i+1} \times 4) \bmod 64))$
   end if
   $o_{i+3}$ = '='
end if
```

If the length of input in byte is not divisible by three, then the special padding character '=' is needed

Background (High-level Synthesis)

- For developers, researchers, and scientists
  - Little hardware development experience
  - Take advantage of the potential benefits of FPGA-based heterogeneous computing systems
- OpenCL Application (https://www.khronos.org/opencl/)
  - Host (C, C++, Boost, PyOpenCL)
  - Kernel (OpenCL, OpenCL C++)
- Portability
  - Program (OpenCL 1.2 and part of OpenCL 2.0+)
  - Performance (Platform-dependent)

Streaming Interface

- CUDA Stream
  - An interface for overlapping data transfers and kernel computations
  - Users need to decompose a problem space
  - A proprietary API works only on NVIDIA GPUs

- OpenCL Stream
  - No API available
  - Realized with multiple command queues and domain decomposition
  - An open standard for writing portable programs on different platforms

Base64 Encoding Kernel in CUDA C/C++

```
__constant__ uchar T[64] = “ABCD...789+/”;      // 64-entry look-up table
__global__ void
base64_enc ( const uchar*__restrict__ input,
                  uchar*__restrict__ output,
             const   char padCount,             // the number of paddings for the last 3-byte group
             const size_t numBlock,             // input size in a group of three bytes
             const    int offset )              // thread offset
{
 size_t id = offset + blockDim.x * blockIdx.x + threadIdx.x;
 if ( id >= numBlock ) return;
 bool last = (padCount != 0) & (id == numBlock-1);
 int tid = id * 3;
 int otid = id * 4;
 uchar si  = input[tid];
 uchar si1 = input[tid+1];
 uchar si2 = input[tid+2];
 uchar r0 = T[si / 4];
 uchar r1 = (last && padCount == 1) ?
            T[(si * 16) % 64] :
            T[(si * 16) % 64 + si1 / 16];
 uchar r2 = last ? ((padCount == 1) ? '=' : T[(si1 * 4) % 64]) :
                                       T[(si1 * 4) % 64 + si2 / 64];
 uchar r3 = last ? '=' : T[si2 % 64];
 out[otid  ] = r0;
 out[otid+1] = r1;
 out[otid+2] = r2;
 out[otid+3] = r3
}
```

Compute thread ID and workload

Load the 3-byte data

Base64 Encoding

Store the 4-byte results

Base64 Encoding Kernel in CUDA C/C++ (Vectorized Memory Accesses)

```
__global__
void  base64_enc ( … )
{
   … …
   uchar3 s = ((uchar3*)input)[id];
   uchar4 r;
   r.x = T[s.x / 4];
   r.y = (last && padCount == 1) ?
         T[(s.x * 16) % 64] :
         T[(s.x * 16) % 64 + s.y / 16];
   r.z = last ? ((padCount == 1) ?
        '=' : T[(s.y * 4) % 64]) :
         T[(s.y * 4) % 64 + s.z / 64];
   r.w = last ? '=' : T[s.z % 64];
   ((uchar4*)out)[id] = r;
}
```

Base64 Encoding Kernel in OpenCL for GPU

```
__kernel void
base64_enc ( __global const uchar*restrict input,
             __global       uchar*restrict output,
                      const  char   padCount,
                      const ulong   numBlock,
                      const   int   offset )
{
  size_t id = offset + get_global_id(0);
  if ( id >= numBlock ) return;
  const uchar T[] = "ABCD...789+/";
  bool last = (padCount != 0) &
              (id == numBlock -1);
 uchar3 s = vload3(id, input);
 uchar4 r;
  r.x = T[s.x / 4];
  r.y = (last && padCount == 1) ?
        T[(s.x * 16) % 64] :
        T[(s.x * 16) % 64 + s.y / 16];
  r.z = last ? ((padCount == 1) ?
        '=' : T[(s.y * 4) % 64]) :
        T[(s.y * 4) % 64 + s.z / 64];
  r.w = last ? '=' : T[s.z % 64];
  ((__global uchar4*)out)[id] = r;
}
```

OpenCL API function

Infer constant memory space

vload3 not the same as (uchar3*)

Base64 Encoding Kernel in OpenCL for FPGA

- Kernel Vectorization
- Compute-unit duplication

```
__attribute__(( reqd_work_group_size(256,1,1) ))
__attribute__(( num_simd_work_items(SIMD) ))
__attribute__(( num_compute_units(CU) ))

__kernel void
base64_enc ( __global const uchar*restrict input,
             __global       uchar*restrict output,
                      const  char   padCount,
                      const ulong   numBlock,
                      const   int   offset )
{
   …
}
```

Work-group size

# SIMD lanes

# compute units

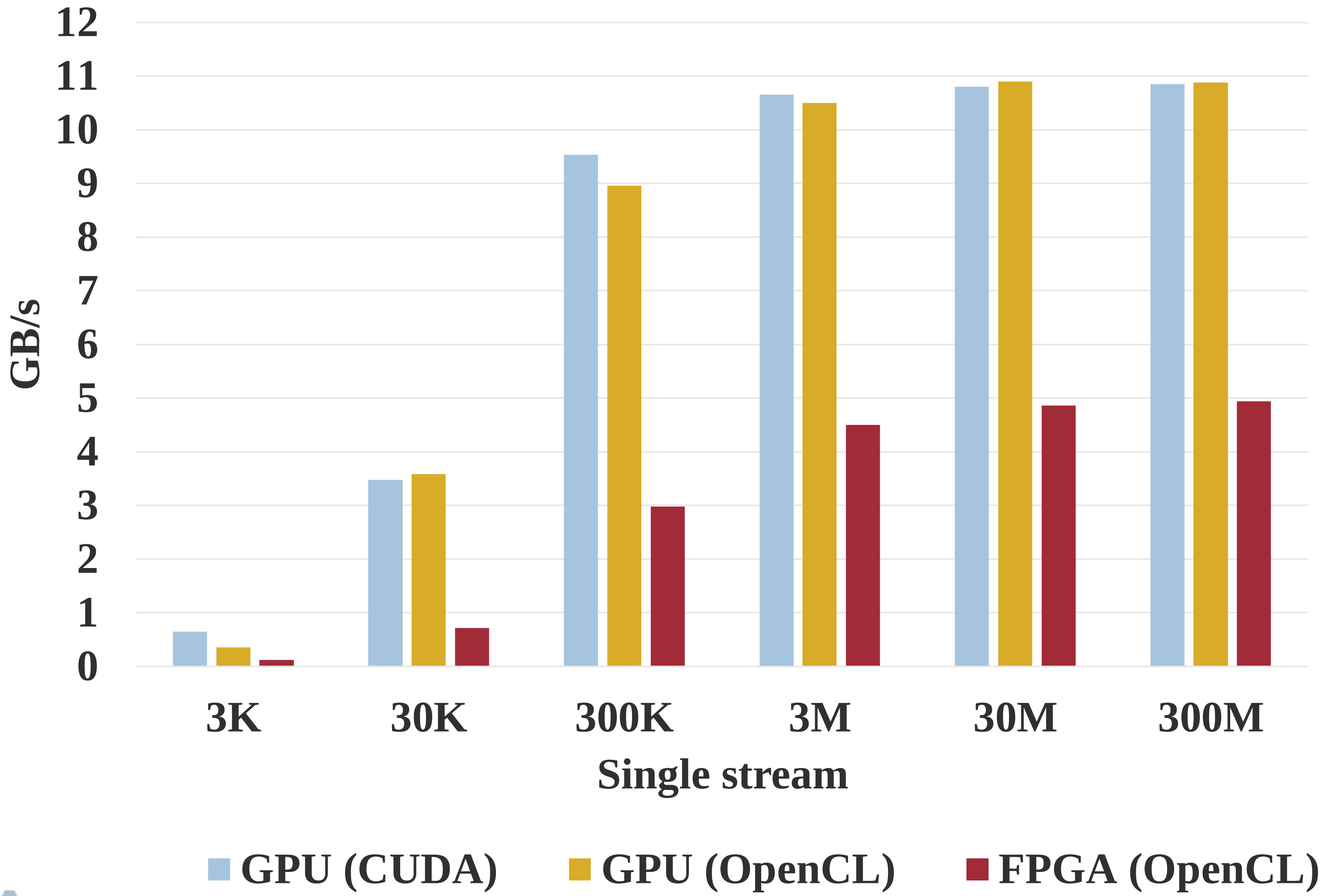
Experimental Setup (Cont.)

- Software development kit
  - Intel FPGA SDK for OpenCL, version 16.0.2 Pro Prime
  - CUDA toolkit, version 10.0.130
- Hardware
  - A Nallatech 385A FPGA card
    - Intel Arria 10 GX1150 FPGA
    - 512-bit memory bus between memory controller and user logic
    - Theoretical memory bandwidth is **34.1** GB/s
  - An NVIDIA P100 GPU
    - 3,584 cores, and theoretical memory bandwidth is **732** GB/s
  - Host
    - The FPGA and GPU are installed on two servers
    - The CPU is Intel Xeon E5-2687W

Experimental Setup

- Input data sizes are 3K, 30K, 300K, 3M, 30M, and 300M
- Run each test case for longer than one second
- $\text{Throughput} = \frac{\#Bytes\ (read) + \#Bytes\ (write)}{Kernel\ offloading\ time}$
- $\text{Speedup} = \frac{Tb}{Ts}$
- $\text{Efficiency} = \frac{Speedup_{max}}{nStreams}$

Baseline Performance on the GPU and FPGA



Performance (GB/s) of Vectorized and Scalar Memory Accesses on the GPU

Input size	GPU scalar	GPU vectorized
3K	0.64	0.65
30K	3.48	3.47
300K	9.54	9.29
3M	10.66	10.59
30M	10.84	10.80
300M	10.85	10.81

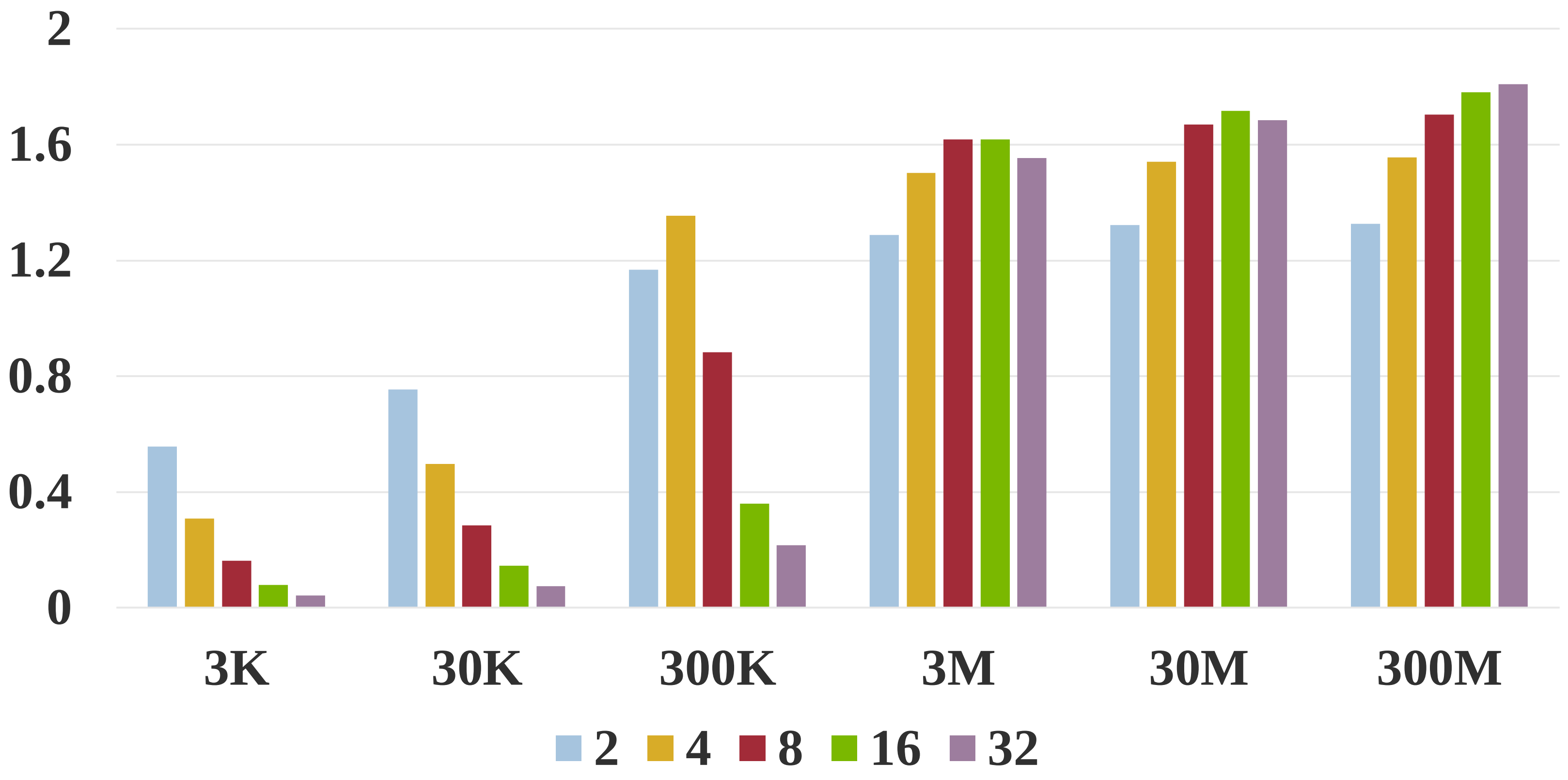
Vector types in OpenCL or CUDA on the GPU is not for performance, and more work-items is better than large vectors per work-item

Performance (GB/s) of Vectorized and Scalar Memory Accesses on the FPGA

Input size	FPGA SIMD1	FPGA SIMD16	
3K	0.1	0.1	
30K	0.51	0.66	
300K	1.25	2.86	
3M	1.44	4.17	
30M	1.49	4.61	
300M	1.49	4.69	3X

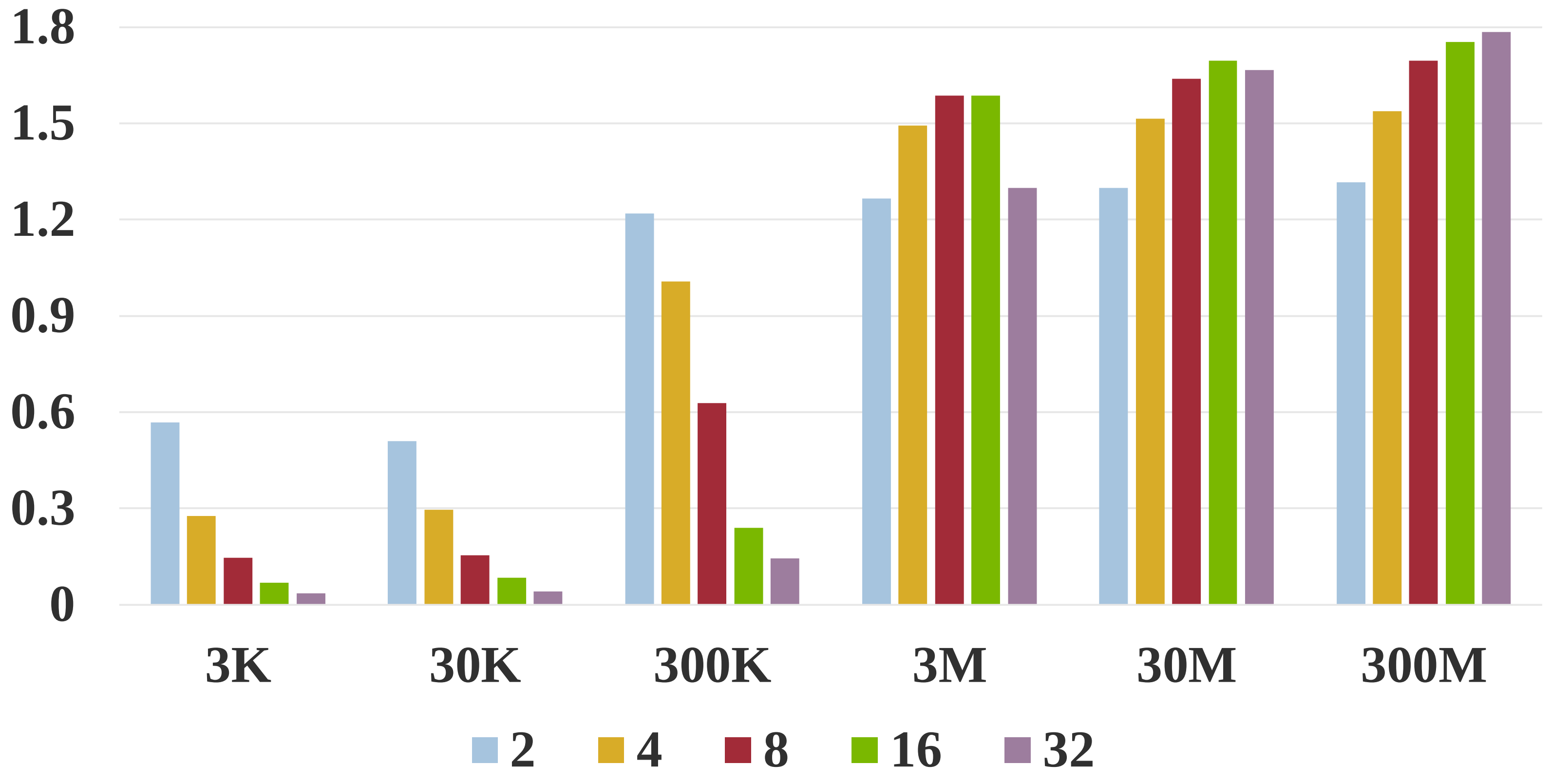
Leverage the 512-bit memory bus between user logics and the memory controller to achieve higher utilization of memory bandwidth

Performance Speedup and Efficiency using CUDA Streams on the GPU



#Streams	2	4	8	16	32
Speedup	1.32	1.55	1.7	1.78	1.81
Efficiency	66.3%	38.9%	21.3%	11.1%	5.7%

Performance Speedup and Efficiency using OpenCL Streams on the GPU

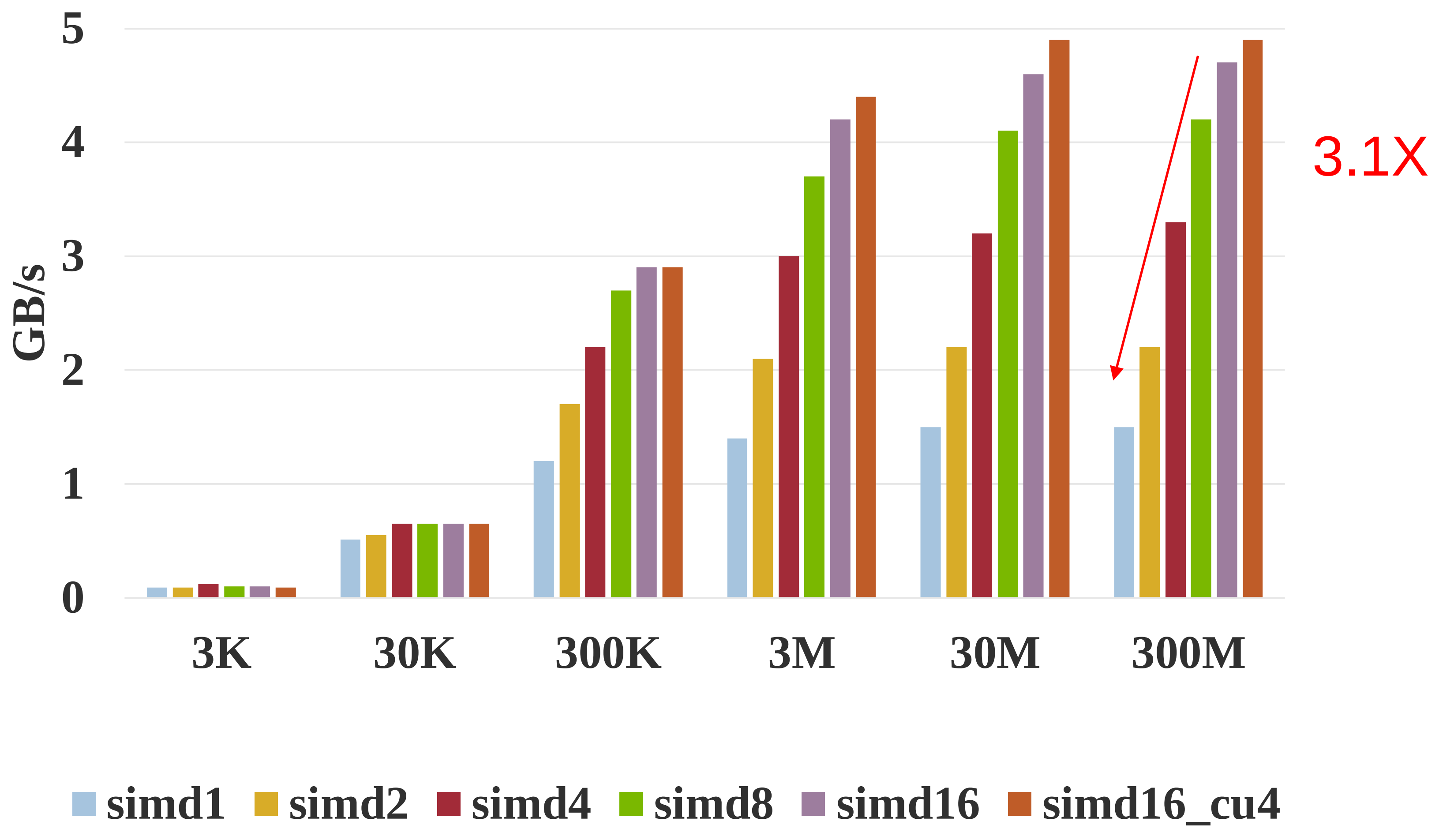


#Streams	2	4	8	16	32
Speedup	1.32	1.54	1.69	1.75	1.78
Efficiency	65.8%	38.5%	21.2%	10.9%	5.5%

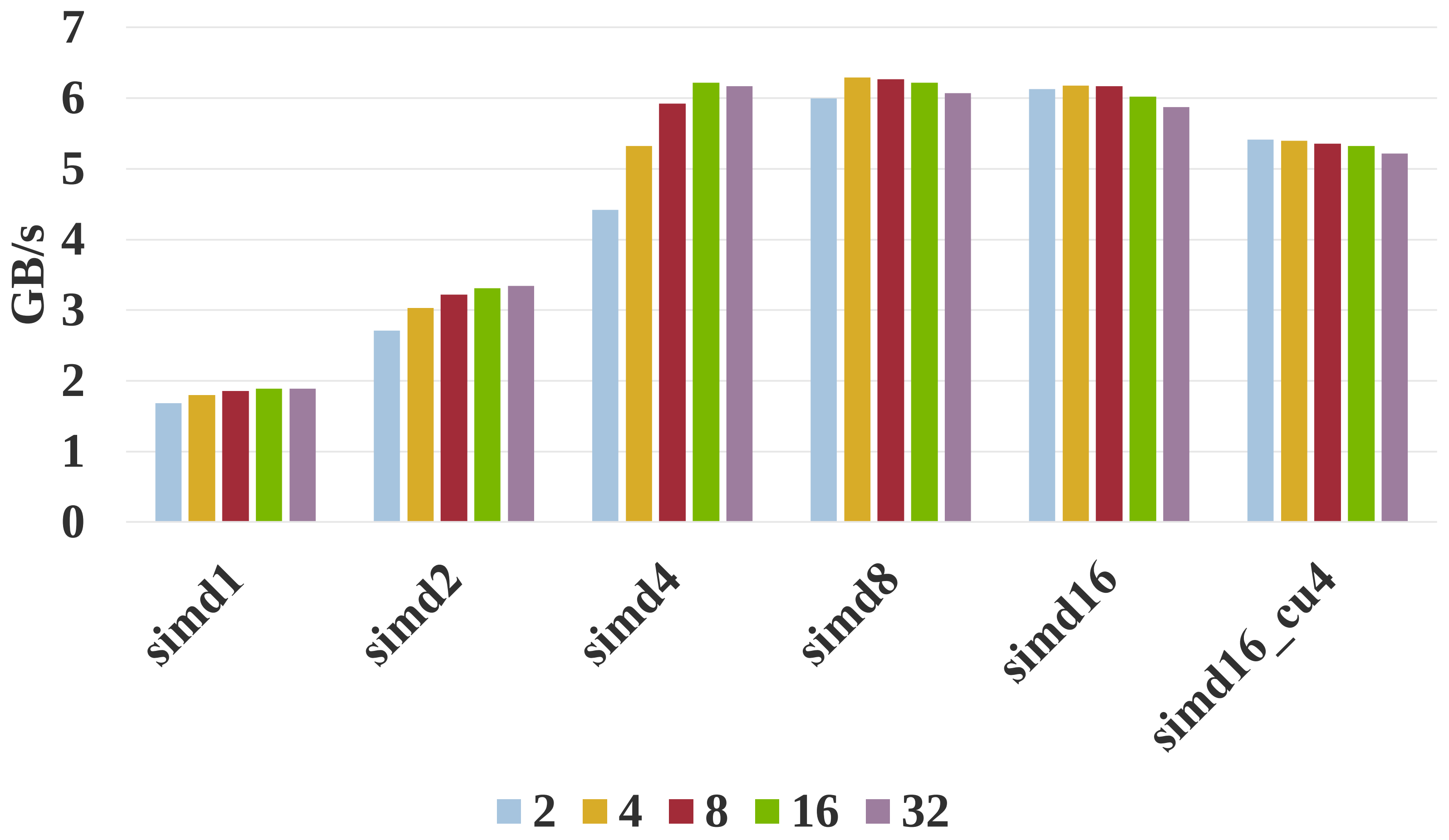
Resource Utilization on the Intel Arria10 GX1150 FPGA

Kernel	Logic utilization	Memory blocks utilization	Fmax (MHz)
simd1	12%	11%	292
simd2	13%	12%	272
simd4	13%	12%	288
simd8	13%	12%	291
simd16	14%	12%	272
simd16_cu4	22%	19%	195

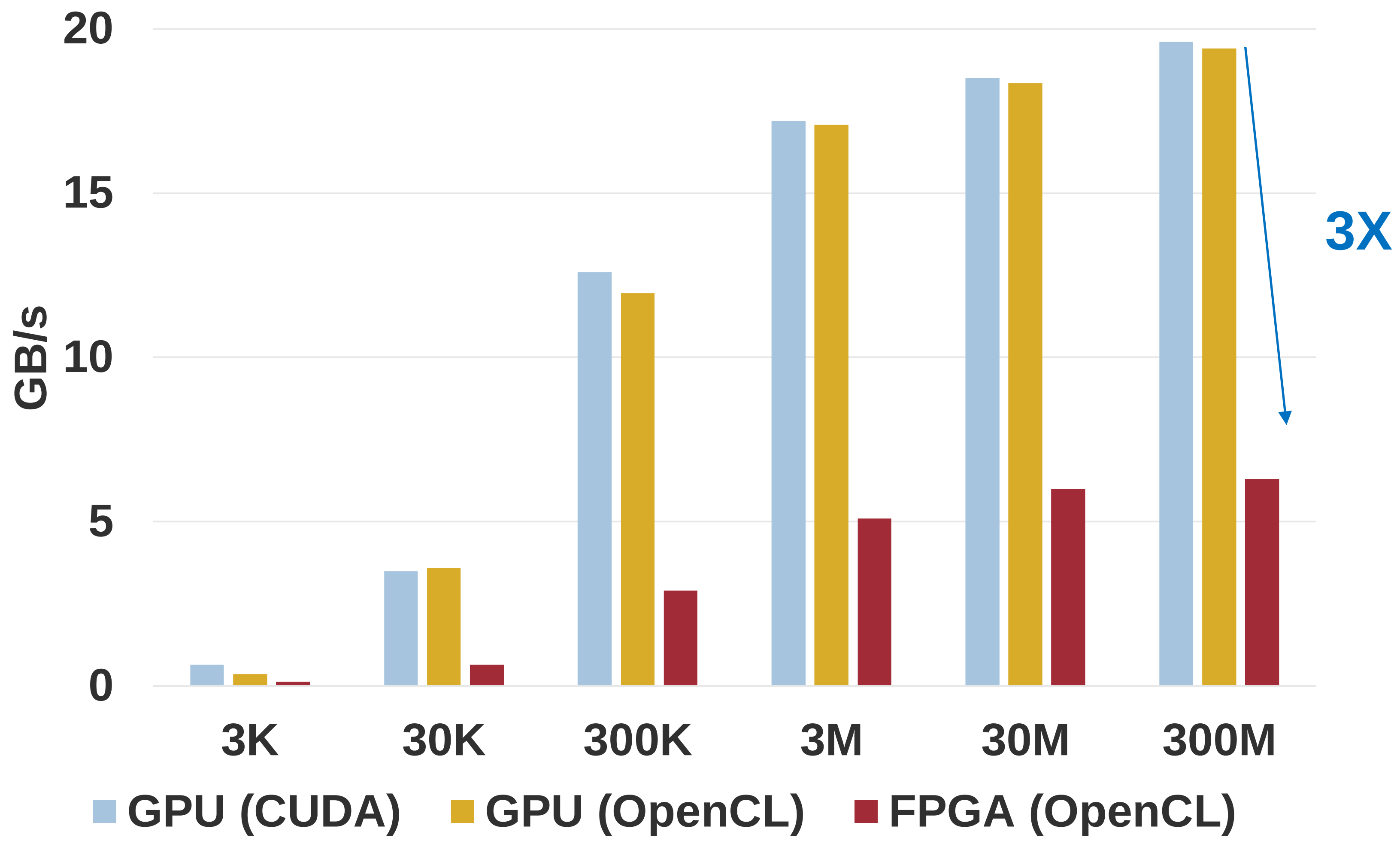
Baseline Kernel Throughput with respect to the SIMD Width and Input Size on the FPGA



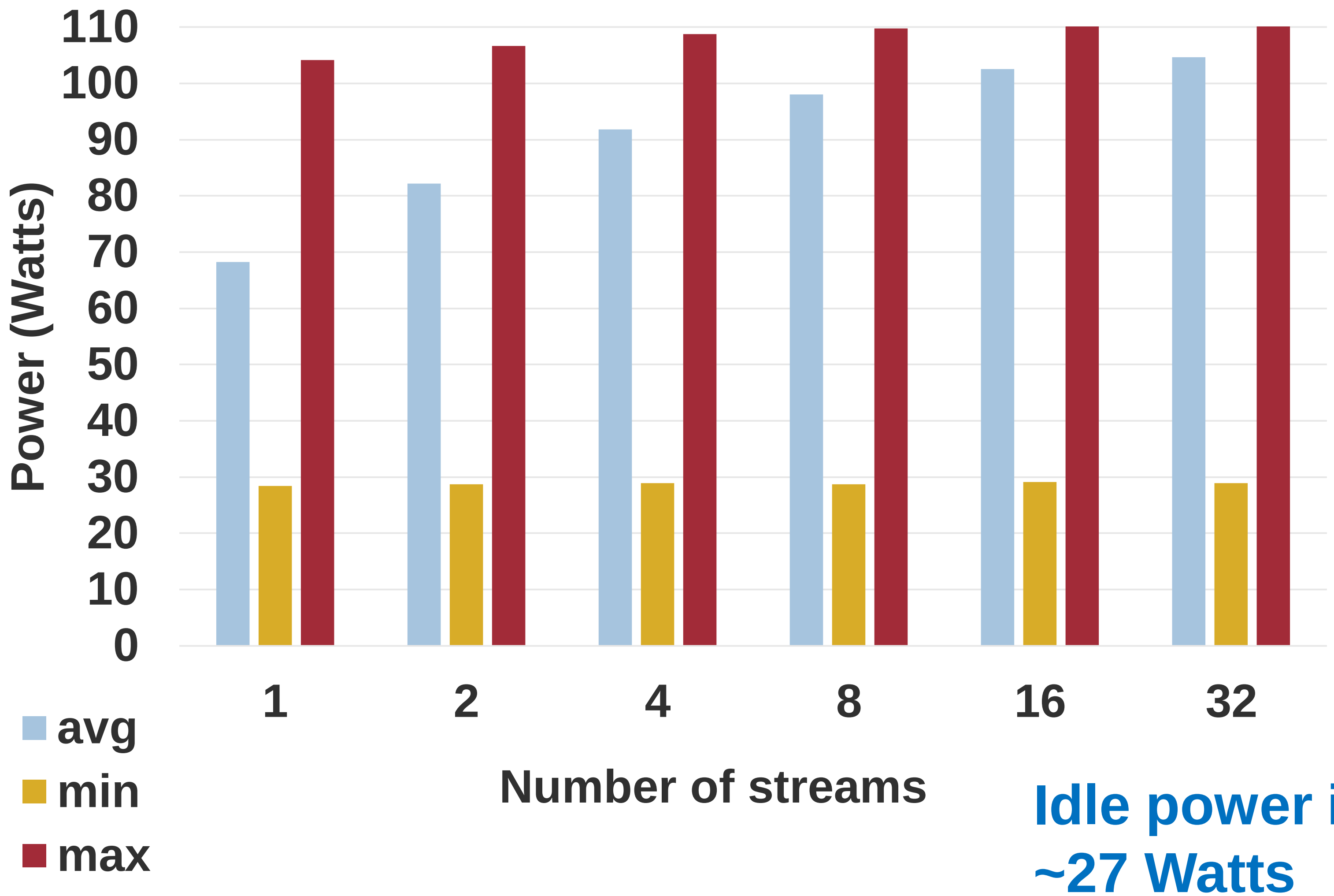
Kernel Throughput with respect to SIMD Width and Stream Count on the FPGA (Input size is 300M)



Optimized Kernel Throughput on the GPU and FPGA

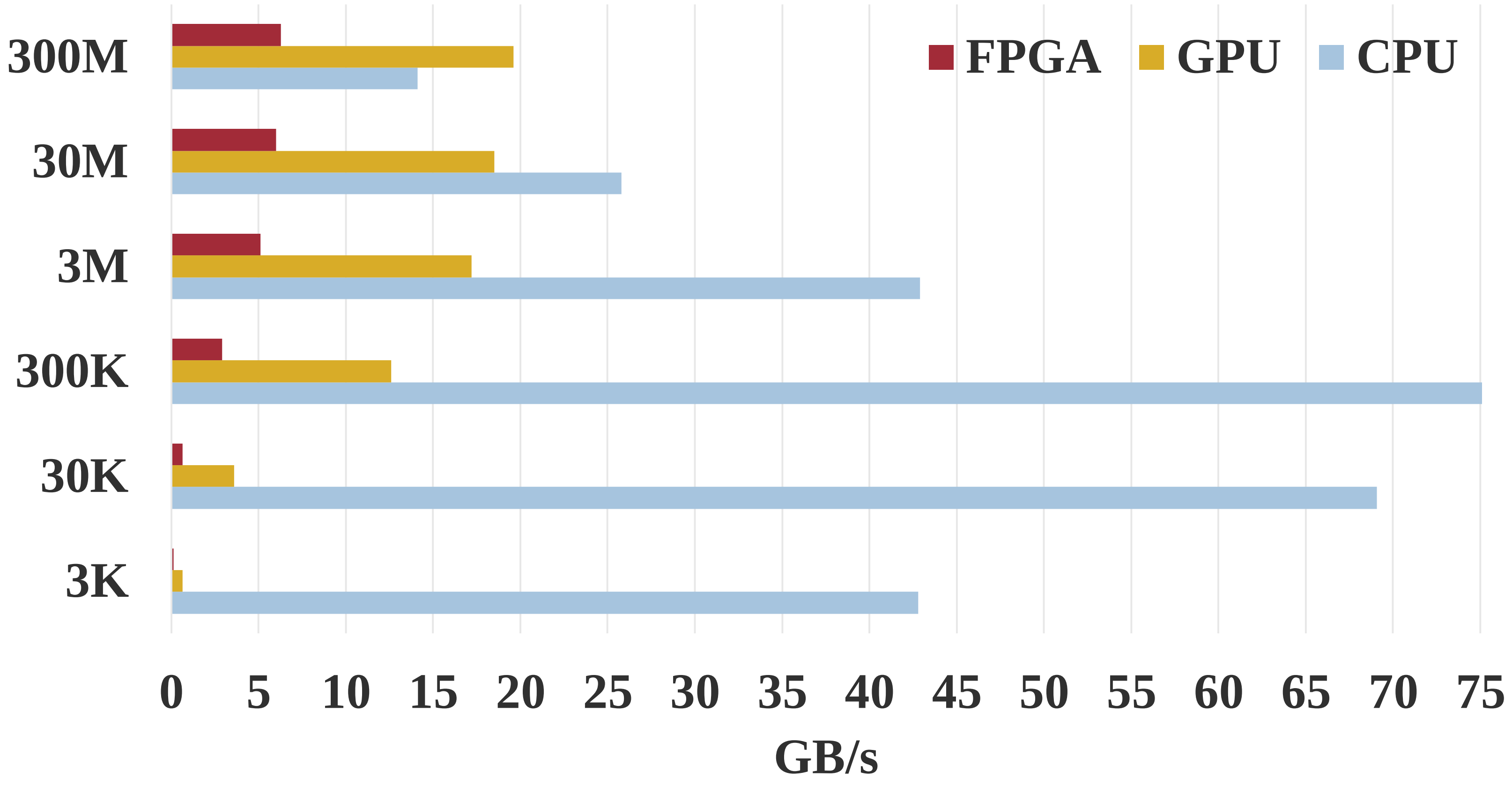


The Minimum, Average, And Maximum Power Consumption on the GPU. The input size is 300M.



Idle power is ~27 Watts

Performance comparison on the Nallatech 385A FPGA card, NVIDIA P100 GPU, and Intel Xeon Platinum 8180 CPU



https://github.com/lemire/fastbase64

Conclusion

- FPGA and GPU have fundamentally different computing architectures
- CUDA and OpenCL streams can improve the application performance.
- But they should be used with caution when decomposition does not maintain all hardware resources of a GPU busy
- FPGAs are promising low-power heterogeneous computing component
- There is increasing benefit of offloading the kernel computation

Thanks