University of Pennsylvania Department of Electrical and System Engineering System-on-a-Chip Architecture

ESE5320, Fall 2023 Design and Function Milestone Wednesday, October 30

Due: Friday, Nov. 8, 5:00PM

Group: Develop functional code. Identify design space options. Writeup (single turn-in for group).

- 1. Identify major design space axes that could be explored for your implementation.
 - For this milestone, aim for breadth (quantity of options)
 - Each axis description can be 1–3 sentences. Identify challenge being addressed, basic solution opportunity, and continuum. A single point in the design space is not a continuum; except in rare cases, this should capture a range of potential parameter values.
 - Include a simple equation to illustrate ideal benefit (e.g., running N tasks in parallel reduces runtime by a factor of N; T(N) = T(1)/N) and the associated resource costs.
 - Cover all operations except SHA that must be accelerated including communication among operators. (i.e., CDC, Deduplication, LZW, and communication/integration)
 - You have four top-level options for SHA as noted on the next page which will make some of your primary design choices there, so we do not ask you to detail SHA here, but you should include communication with the SHA operations.
 - Aim for at least 6 axes per operation. Identify a few associated with how operations interact with each other.
 - Some of this should build on the parallelism opportunities you identified on the previous milestone.

Axis:	P, number of butterfly units.
Challenge:	Improving the throughput of the FFT
Opportunity:	Implement multiple hardware butterfly datapath units.
Continuum:	This can range from 1 to a fully spatial design with $P =$
	$\frac{N}{2}\log(N)$ butterfly units.
Equation for Benefit:	$\overline{T}hroughtput(P) = P \times SingleButterflyThroughput$
Equation for Resources:	$Resources(P) = P \times SingleButterflyResources$

Example from FFT design discussed in class.

- 2. Refine your placeholder implementation into a functional implementation for the project task that can run on a Zynq ARM Cortex A53 processor and produce a valid compressed output stream that works with the supplied decompressor. Integrate with the provided ethernet input flow. Compress from ethernet input to SDCard output.
 - The primary goal for this assignment is functionality. As such, you should focus on a simple design that captures the necessary behavior.
 - As a result, this design need not be efficiently synthesizable to hardware.
 - However, you will eventually be optimizing this design and likely exploring HLS mappings to hardware. So, given a choice, you might want to use design constructs and idioms that you know will be more amenable to HLS hardware mappings.
 - Alternately, you should be prepared to rewrite your code later for efficient hardware mappings.
- 3. Turn in a tar or zip file with your functional code to the designated assignment component in canvas.
- 4. Turn in a tar or zip file with binaries to support execution of your code to the designated assignment component in canvas.
 - (a) The tar (or zip) files should include:
 - encoder binary for your encoder to run on the the Ultra96
 - (b) Your encoder should take one argument:
 - the file name where the program should store the compressed data.
- 5. Measure the raw ethernet performance from your host machine to your Ultra96 (See Section 13.1 in Project Handout).
- 6. Document your design.
 - (a) Code sources (e.g., URLs) for any open-source code you used as a starting point or as a primary reference
 - (b) Current compression ratio and breakdown of contribution from deduplication and from LZW compression.
 - (c) Overall throughput (Gb/s) of your current implementation.
 - (d) Description of all validation performed on your current functional implementation.
 - (e) Report the raw ethernet speed measurements (Problem 5).
 - (f) Description of who did what. How did your team collaborate on the design, implementation, and validation?

- 7. Identify any challenges your group had in collaboration and design integration this week and how you plan to address them for future weeks.
 - This is not a question about technical status that should be addressed above.
 - This is for teamwork, coordination, collaboration, communication, and workflow issues.
 - These may be things you've overcome by submission but didn't go as smoothly as they should have.
 - In the unlikely case that everything went perfectly, identify the things you did that made it work well. For your future plans, look forward to next week to see if the same techniques are applicable or if there are new challenges that might require different or additional techniques for things to continue to go well.

SHA

You have four options for implementing SHA in your encoder pipeline:

- 1. Writing serialized SHA-256 or SHA3-384 running on the ARM processor.
- 2. Using the dedicated SHA3-384 unit in the Zynq Ultrascale.
- 3. Using SHA3-384 NEON intrinsics.
- 4. Implementing SHA-256 or SHA3-384 on FPGA using HLS (not until P3).

The specific option you choose will have implications on the maximum throughput you can achieve for your encoder pipeline. Moreover, you will need to adjust your design accordingly if you end up using the SHA3 unit, since it has a different digest size.

- You can find throughput comparisons of SHA3-384 on NEON and the dedicated SHA3 unit here: https://www.xilinx.com/support/documentation/white_papers/wp512-accel-cryppdf.
- You can find an example of how to use the SHA3 unit here: https://xilinx-wiki. atlassian.net/wiki/spaces/A/pages/18841654/Linux+SHA+Driver+for+Zynq+Ultrascale+ MPSoC Note we have already enabled the SHA3 unit and user-space API in a provided platform. You will just need to write the driver.
 - N.B. To use this unit, data provided must be padded to a multiple of 4B to get consistent results; providing this padding and extracting full performance can be tricky.
- You can find software implementations that use NEON intrinsics here:
 - https://github.com/james-ben/mpsoc-crypto
 - https://github.com/noloader/SHA-Intrinsics

Multiple Cores

Recall from Homework 3 that you can utilize multiple cores using std::threads. There are four cores (ARM Cortex-A53) on the Ultra96. Moreover, recall that each core on the Ultra96 has one 64-bit NEON SIMD unit with 128-bit registers that you can utilize simultaneously with std::threads.

FPGA Acceleration Tutorial: Bloom Filter

Using bloom filter as the application, this tutorial shows you:

- a significant speedup $(8\times)$ when computations are offloaded on the FPGA efficiently.
- how to write an HLS kernel for a CPU implementation (using ap_uint, hls::stream, and pragmas).
- how to achieve communication-compute overlap using sub-buffers.
- 1. Clone the ese532_code repository using the following command:

git clone https://github.com/icgrp/ese532_code.git

If you already have it cloned, pull in the latest changes using:

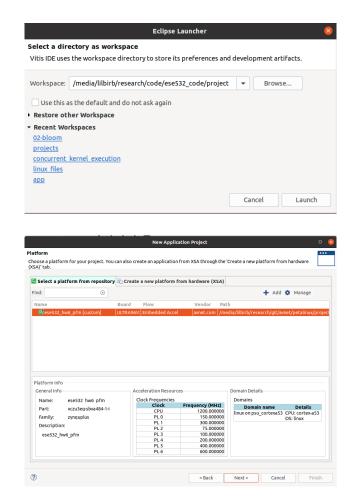
cd ese532_code/ git pull origin master

The code you will use for this section is in the **vitis_tutorials/bloom** directory. The directory structure looks like this:

```
bloom/
cpu/
fpga/
MurmurHash2.c
common.h
compute_score_fpga_kernel.cpp
compute_score_host.cpp
hls_stream_utils.h
main.cpp
sizes.h
xcl2.cpp
xcl2.hpp
```

The cpu folder has a standalone CPU implementation of the bloom filter, which you can compile using the Vitis GUI flow from P2 and run it. The main.cpp code in the fpga folder has the OpenCL host code. The top level HLS function is in compute_score_fpga_kernel.cpp. We will now show how to use the Vitis GUI flow to compile OpenCL and HLS code.

2. Create or use an existing workspace, create a new application project and use the provided platform.



3. Choose Empty Application from the SW acceleration templates as follows:

		New Applica	tion Project			0 😣
Templates Select a template to create your	project.					
Available Templates:						
Find:	٥	⊟ ⊞	Empty Application			
 SW acceleration templates Empty Application 			Creates a new Emp	cy application		
Vector Addition ▼ SW development templates Empty Application (C++) Linux Empty Application Linux Hello World	Libraries					
?			< Back	Next >	Cancel	Finish

4. Right-click on the **src** folder and click on import sources. Import the source files for this project as follows:

Import Sources	o 😣
File system Import resources from the local file system.	
From directory: /media/lilbirb/research/code/ese532_code/vitis_tutorials/bl v	Browse
✓ & fpga ✓ & fpga ✓ & formation for the second	
Filter Types Select All Deselect All	
Into folder: bloom/src	Browse
Options Overwrite existing resources without warning Create top-level folder Advanced >>	
Cancel	Finish

5. Click on bloom.prj from the Explorer and change the Active build configuration to Hardware as shown below:

	vitis_workspace - bloom/blo	om.prj - Vitis IDE	
<u>File Edit</u> Search Xilinx Project Window Help			
🗂 • 🗟 🐚 🖲 • 🔦 • 🎋 • O • 🔗 • 🖾 🖉 🖬	1 to φ + φ +	Q	uick Access
🖳 Explorer 🛛 📄 🗢 🔻 🗖 🗖	🛎 bloom_system 🛛 🛠 bloom 🛙	- 8	🗄 Outlin
▼ m > bloom_system [ese532_code master] [ese532_hw6_pfm] ▼ ⊕ > bloom [ese532_code master] [linux on psu_cortexa53]	X Application Project Settings	Active build configuration: Hardware	Anoutline
► 🔊 Includes	General	Options	
► 🔐 > Emulation-HW	Project name: bloom	Target: Hardware	
 ▶ ₽² > Emulation-SW ▶ ₽² > Hardware 	Platform: ese532 hw6 pfm	Host debug:	
ler > src	Runtime: OpenCL	Kernel debug:	
👔 bloom.prj	Number of devices: 1 – +	Kernel debug mode: 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸 🗸	
s, bloom system.spri		Report level:	
		Hardware optimization: Default optimization (-00)	
	Hardware Functions	E 🗄 🖉 🗶 👗	
	Name Compute Units Port Data Width	Max Memory Ports	
Assistant 22 A			
	📮 Console 🛿 🖺 Problems 🗐 Vitis Log 🛈 Guidance 🛛 🐥 😚 😫	🔄 📓 🖻 🖳 🖻 🗣 📑 🖛 🧧 🖬 🗐	

6. Add an xclbin container by clicking on the button circled in red below. Rename the

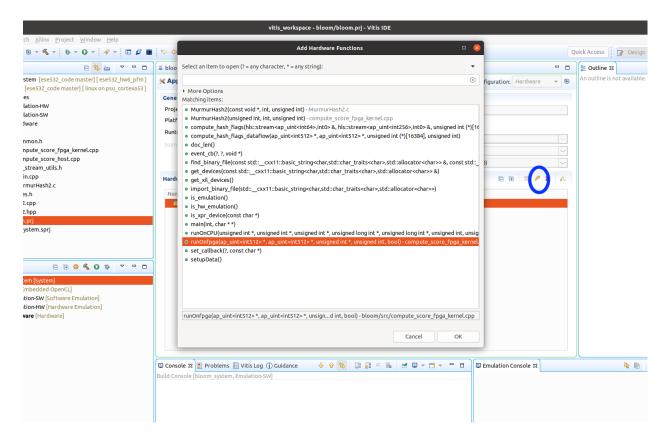
container to runOnfpga_hw by clicking on the name binary_container_1. This is the xclbin name that the host code uses.

	■ \$P\$ \$P\$ \$P\$ \$P\$			Ç	uick Access	Design	🌾 Debi
Explorer 🛛 🕒 🗢 🖻	🗆 🛎 bloom_system 🛛 🛠 bloom 😫 🖻 main.cpp			- 8	E Outline 🛙		•
> bloom_system [ese532_code master] [ese532_hw6_pfm			Active build configuration: Hardware	- 19	An outline is no	t available.	
▶	General	Options					
Emulation-HW	Project name: bloom	Target:	Hardware				
 ▶ @‡ > Emulation-SW ▶ @‡ > Hardware 	Platform: ese532 hw6 pfm	Host debug:					
+ @ ⁺ > SFC	Runtime: OpenCL	Kernel debug:					
► 🞼 common.h	Number of devices: 1 – +	Kernel debug mode:	Waveform	\sim			
 		Report level:	Default	\sim			
 is compute_score_nosc.cpp is hls_stream_utils.h 		Hardware optimization	n: Default optimization (-O0)	\sim			
► 🞼 main.cpp	Hardware Functions		= = () :	AX			
MurmurHash2.c	for such the last back with the						
▶ ፼ sizes.h ▶ ፼ xcl2.cpp	Name Compute Units Port Data Width		Max Memory Ports				
Assistant 22 Bloom_system [System] © bloom [Embedded OpenC] + $\langle Emulation SW [Software Emulation] + \langle Emulation SW [Vardware Emulation] + \langle Hardware [Vardware]$							
							0

7. Now click on the button circled in blue below. Choose the **runOnfpga** function as the hardware function.

ESE5320

Fall 2023



8. From the Assistant view, double click on Hardware as follows:

占 Explorer 🕱 📄 🗢 🗖 🗖	🛎 bloom_system 🛛 🔀 bloom 🕱 🗟 main.cpp	
r > bloom_system [ese532_code master] [ese532_hw6_pfm]	Application Project Settings	
<pre>- @ > bloom [ese532_code master] [linux on psu_cortexa53]</pre>	General Project name: bloom Platform: ese532 hw6 pfm Runtime: OpenCL Number of devices: 1 Hardware Functions I Name Compute Units Port Data Widte ✓ ImunOnfpga_hw ✓ runOnfpga 1 Automatic	- + Hos Kerr Rep Harr
Assistant X Bioom_system [System] Cip bloom [Embedded OpenCL] Cip K Emulation-SW [Software Emulation] Cip K Emulation SW [Software Emulation] Cip K Hardware [Hardware] Cip K Hardware [Hardware]		

9. Click on Hardware→runOnfpga_hw→runOnfpga. This brings the screen where you can specify the number of compute units, compiler options for the kernel, assign different ports to inputs etc. Keep the defaults for now:

					Ha	rdware Function	Settings			o 😣
type filter t 🛞 🖻 🖿	🕖 runOnfpga									⇔ • ⇔ • •
✓ ➡ bloom_system ✓ ➡ bloom ♦ < Emulation-SW	Name: Compute units:	runOnfpga								
 Kemulation-SW Kemulation-HW 	Max memory ports:									
→ → Hardware	Port data width:	Auto 👻								
✓	Extra source files:									
	V++ compiler options:									
	Compute Unit Settings									
	Name	Memory Auto	SLR	ChipScope Debug	Protocol Checker	Data Transfer None	Execute Profiling	Stall Profiling		
	 funOnfpga line runOnfpga_1 		Auto			None				
	output_flag	gs Auto				None				
	input_word					None				
	doom_filte					None				
	load_filter									
	Refresh V++ Compiler Comma									
	<pre>\$(XILINX_VITIS),</pre>	-config.cfg pga_hw-runOnfpg;	a-compi ga.xo"	le.cfg "/src/compute_sc	ore_fpga_kernel.cp	-			Revert	Apply
									Cancel	Apply and Close

10. Click on bloom.prj. Check out the Hardware optimization option where you can change the optimization level for the hardware function. Additionally, recall from P2 that you can change the optimization level of the host code from the C/C++ build settings. Now click on the build button on the menu bar to start compilation:

	vitis_workspace - bl	loom/bloom.prj - Vitis IDE		- • 8
Eile Edit Search Xilinx Project Window Help			a da ta	
			Quick Acc	
Explorer Explorer		e_score_fpga_kernel.cpp		tline ⊠ □ □
 Bloom_system [eses32_code master] [eses32_nwo_prm] bloom [ese532_code master] [linux on psu_cortexa53] 	X Application Project Settings	Active build configuration:	Hardware 🔻 🛞 An out	tine is not available.
▶	General	Options		
▶ @÷ > Emulation-HW	Project name: bloom	Target: Hardware		
▶ Q ⁴ > Emulation-SW ▶ Q ⁴ > Hardware	Platform: ese532 hw6 pfm	Host debug:		
	Runtime: OpenCL	Kernel debug:		
▶ Is common.h	Number of devices: 1	Kernel debug mode: Waveform	<u></u>	
Compute_score_fpga_kernel.cpp		Report level: Default	~	
 G compute_score_host.cpp G hls stream utils.h 		Hardware optimi ation: Default optimization (-00)		
Fights_stream_outs.m	Hardware Functions		EPX A	
MurmurHash2.c	nardware runctions	6.6		
▶ I sizes.h	Name Compute Units Port Data Wi	dth Max Memory Ports		
 ▶ G xcl2.cpp ▶ G xcl2.hpp 		Auto		
% bloom.prj	📝 runOnfpga 1 刘	KUCO		
& bloom_system.sprj				
🖌 Assistant 🛛 📄 🗄 🕸 🔦 🗿 🌞 🔻 🗖 🗖				
→ [™] bloom_system [System]				
Generation (Embedded OpenCL) Software Emulation)				
Kenulation-HW [Hardware Emulation]				
Hardware [Hardware]				
		🖟 😚 💷 🔐 두 🐘 📑 💷 🕶 🗂 🕶 🗖 🖬 🖾 🖬 🖬	(
	Console 22 Problems 2 Vitis Log (i) Guidance Build Console [bloom, Hardware]	▷ 수 🧐 📰 두 🐘 📑 🔍 ▾ 🗂 🗖 Emulation Co	Shsole 🛛	k 🗎 🗸 🗖
	build Console [bloom, Hardware]			

11. Once the compilation completes, open the Hardware folder from the Explorer. The binaries are in the package/sd_card folder.

🖥 Explorer 🛿 📄 🔽 🔻 🗖	🗖 🔏 bloom_system 🛛 🔀 bloom 🛿 🗟 main.cpp 🔹 com
✓ □ > bloom_system [ese532_code master] [ese532_hw6_pf ✓ □ > bloom [ese532_code master] [linux on psu_cortexa!	m] 🔀 Application Project Settings
Gill Includes	General
Emulation-HW	
Emulation-SW	Project name: <u>bloom</u>
✓	Platform: <u>ese532 hw6 pfm</u> ···
🕶 😅 package	Runtime: OpenCL
✓ ≝ sd_card	Number of devices: 1
📄 bloom	
BOOT.BIN	
boot.scr	
📄 image.ub	Hardware Functions
init.sh	Name Compute Units Port Data
platform_desc.txt	
📄 runOnfpga_hw.xclbin 📄 system.dtb	✓
	📕 runOnfpga 1
ese532 hw6 pfm.bif	
sd_card.img	
P ackage.build	
k 🛋 runOnfoga, hw.build	
🖌 Assistant 🛿 📄 🖻 🕀 🗳 🤝 🖓	
 Bloom_system [System] Bloom [Embedded OpenCL] Semulation-SW [Software Emulation] Emulation-HW [Hardware Emulation] Hardware [Hardware] 	

12. Copy the binaries and the **xrt.ini** to the Ultra96 as follows and then reboot the Ultra96.

$\langle \rangle$			D r	esearch code	ese532_cod	e vitis_work	space bloom	Hardware	package	sd_card 👻	· Q			≣▼≡	- • ×
③ Recent	\bigcirc	0100 0010 1001 0110	D100 D010 1001 0110	0100 0010 1001 0110	>		0100 0010 1001 0110	D100 D010 1001 0110							
★ Starred	bloom	boot.scr	BOOT.BIN	image.ub	init.sh	platform_ desc.txt	runOnfpga_ hw.xclbin	system.dtb							
습 Home															
Desktop			шы	ˈb@stingy: /media	/lilbirb/researc	h/code/ese532	_code/vitis_works	pace/bloom/H	ardware/pacl	kage/sd_card			Q = -	8	
				_code/vitis_wor										1	
	7.1's password		100%	6627KB 38.0ME	3/s 00:00										
Picturimage.ub			100%	2007 731.6KE 8117KB 33.0ME code/vitis_wor	3/s 00:00		ackage/sd card	S scp bloom	runOnfpga	hw.xclbin	root@10.10	7.1:~/			
⊟ Videc root@10.10.7 bloom	7.1's password		100%	1045KB 4.5MB	3/s 00:00					_					
Trash runOnfpga_hw lilbirb@stin				5468KB 43.2MB code/vitis_wor				is 🗌							
🖹 resea															
+ Other															

13. Run the code using the following commands in the Ultra96:

ifconfig eth0 10.10.7.1 netmask 255.0.0.0
export XILINX_XRT=/usr
./bloom 40000 64

You should see the following output in the terminal:

```
root@ultra96v2-2020-1:~# ./bloom 40000 64
Initializing data
Creating documents - total size : 559.858 MBytes (139964416 words)
Creating profile weights
[ 1018.547572] [drm] Pid 769 opened device
[ 1018.551450] [drm] Pid 769 closed device
[ 1018.558627] [drm] Pid 769 opened device
Loading runOnfpga_hw.xclbin
[ 1018.617733] [drm] zocl_xclbin_read_axlf The XCLBIN already loaded
[ 1018.617765] [drm] zocl_xclbin_read_axlf 3c650f2f-9cc2-408a-8c92-0ec3bc33
[1018.633496] [drm] bitstream 3c650f2f-9cc2-408a-8c92-0ec3bc335ce3 locked,
[ 1018.641197] [drm] Reconfiguration not supported
[ 1018.652995] [drm] bitstream 3c650f2f-9cc2-408a-8c92-0ec3bc335ce3 unlocke
Processing 559.858 MBytes of data
 Splitting data in 64 sub-buffers of 8.748 MBytes for FPGA processing
                -----
[ooqueu]: Completed buffer migrate
```

```
[ooqueu]: Completed buffer migrate
waiting...
[ooqueu]: Completed buffer migrate
waiting...
[ooqueu]: Completed buffer migrate
waiting...
Executed FPGA accelerated version
                                      1414.5707 ms
                                                      (FPGA 247.878 ms)
```

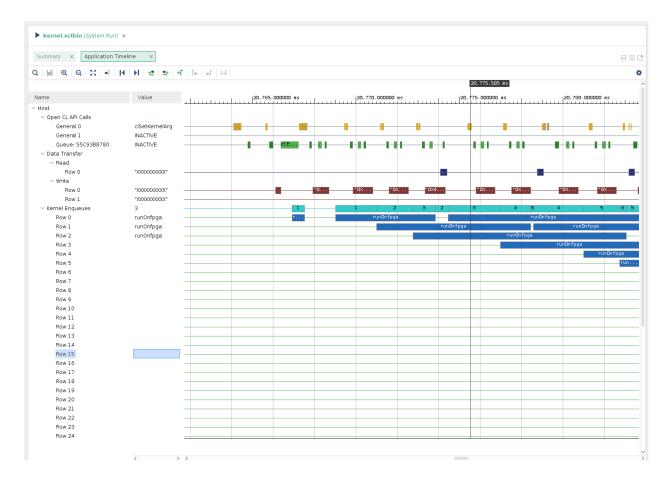
```
14
```

11779.4325 ms

Executed Software-Only version

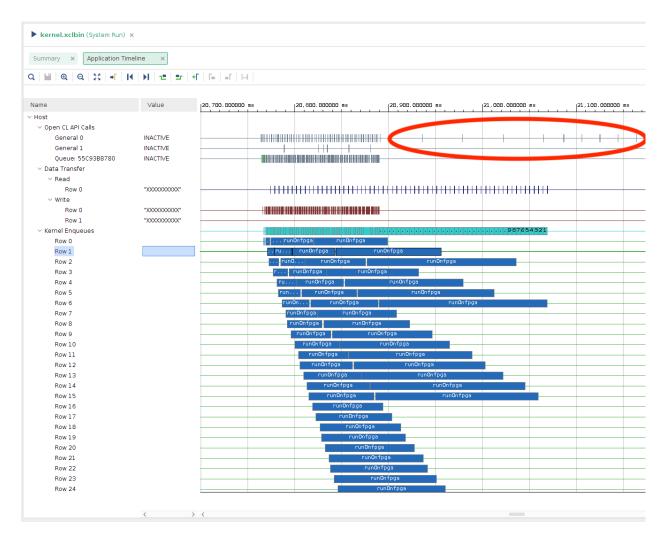
Verification: PASS

- 14. The verbose outputs in the terminal is caused by the call set_callback(flagDone, "oooqueue"); in the host code. This is a helper function in xcl2.hpp that prints out the state of an OpenCL event. You can use it to debug OpenCL calls. In addition, you can also use the OCL_CHECK macro from xcl2.hpp to see if an OpenCL call succeeded.
- 15. Copy the generated run summary and csv files to your host computer and open vitis analyzer. You can see overlap of kernel execution with data transfer and OpenCL API calls.



16. If you scroll forward in the timeline, you can see overlap between computation in the cpu and the fpga as shown below.

Fall 2023



From the output in 13, these calls correspond to the waiting... print outs. You can check in the host code, how we wait for a cl::Event to finish based on a condition, and when the event notifies that it's finished, we start executing the cpu code, so that it overlaps with the fpga execution:

```
needed += size;
if (needed > available) {
   clWaitForEvents(1, (const cl_event *) &flagWait[iter]);
   std::cout << "waiting..." << std::endl;
   available += subbuf_doc_info[iter].size / sizeof(uint);
   iter++;
}
```

17. Now run with a different ITER value and look at the updated trace:

```
./bloom 40000 128
```

Summary × Application Timelin	ie ×																8 0
. ≝ @, Q, X; ≠	N ± ± +																
						2	0,798.1	938 ms									
lame	Value	 20,780.000000	ms	20,790.00000	0 ns		20,8	000000) ms		20,810.00	00000 ms	s	. l	20,820.00	00000 ms	
Host																	
Open CL API Calls																	
General 0	INACTIVE						-									-	
General 1	INACTIVE				_												
Queue: 559FB60780	clEnqueueMigrate							-111	111 - 1			-111			111 1	11 11	
∨ Data Transfer																	
∨ Read																	
Row 0	"X0000000000"									-	1 1	-	-	-			-
V Write					_	_			L .								
Row 0	"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX															-	
Row 1	"X000000000X"																
 Kernel Enqueues 	1					1	1	1 2 3	1 1		2 1 2	- 11	1 2 3			2 1	1 2
Row 0	run0nfpga					Pur	n 0	runt		run0		n0	run		run0.		run0
Row 1					-		r.	Jn0	run0.	**	run0	run	0	runt	D	run0	
Row 2																	
Row 3									-								
Row 4									-								
Device "edge-0"																	

You can see that since the kernel execution time gets smaller as you increase the iteration number, the next kernel execution starts almost immediately.

18. Running a sweep on the number of iterations, we see that ITER=32 is the most performant for this design:

```
./bloom 40000 8
Executed FPGA accelerated version | 1413.3252 ms (FPGA 305.796 ms)
Executed Software-Only version | 11780.3703 ms
_____
Verification: PASS
./bloom 40000 16
Executed FPGA accelerated version | 1396.5072 ms (FPGA 298.034 ms)
Executed Software-Only version | 11770.1858 ms
_____
Verification: PASS
./bloom 40000 32
Executed FPGA accelerated version | 1391.2062 ms (FPGA 284.336 ms)
Executed Software-Only version | 11768.1306 ms
_____
Verification: PASS
./bloom 40000 64
Executed FPGA accelerated version | 1414.5707 ms (FPGA 247.878 ms)
Executed Software-Only version | 11779.4325 ms
_____
Verification: PASS
```

- 19. This concludes a top-down walk-through of this tutorial. To learn more about this design, read the following in-order:
 - (a) Overview of the Original Application
 - (b) Architect a Device-Accelerated Application
 - (c) Implementing the Kernel
 - (d) Data Movement Between the Host and Kernel

Note that the tutorial is written for data center cards. Some of the parameter choices, such as port data width, DDR memory etc. should be reconsidered for the Ultra96 to get optimal performance (refer to this paper: Unexpected Diversity: Quantitative Memory Analysis for Zynq UltraScale+ Systems).

Questions

If anything is unclear please post on Ed Discuss or come to office hours, and we will be glad to assist.