United States Patent	12386779
Kind Code	B2
Date of Patent	August 12, 2025
Inventor(s)	Ray; Joydeep et al.

Dynamic memory reconfiguration

Abstract

Embodiments described herein provide techniques to enable the dynamic reconfiguration of memory on a general-purpose graphics processing unit. One embodiment described herein enables dynamic reconfiguration of cache memory bank assignments based on hardware statistics. One embodiment enables for virtual memory address translation using mixed four kilobyte and sixty-four kilobyte pages within the same page table hierarchy and under the same page directory. One embodiment provides for a graphics processor and associated heterogenous processing system having near and far regions of the same level of a cache hierarchy.

Inventors: **Ray; Joydeep (Folsom, CA), Cooray; Niranjan (Folsom, CA), Maiyuran; Subramaniam (Gold River, CA), Koker; Altug (El Dorado Hills, CA), Surti; Prasoonkumar (Folsom, CA), George; Varghese (Folsom, CA), Andrei; Valentin (San Jose, CA), Appu; Abhishek (El Dorado Hills, CA), Garcia; Guadalupe (Chandler, AZ), K; Pattabhiraman (Bangalore, IN), Kim; Sungye (Folsom, CA), Kumar; Sanjay (Bangalore, IN), Marolia; Pratik (Hillsboro, OR), Ould-Ahmed-Vall; Elmoustapha (Chandler, AZ), Ranganathan; Vasanth (El Dorado Hills, CA), Sadler; William (Folsom, CA), Striramassarma; Lakshminarayanan (Folsom, CA)**

Applicant: **INTEL CORPORATION** (Santa Clara, CA)

Family ID: **1000008748314**

Assignee: **Intel Corporation (Santa Clara, CA)**

Appl. No.: **18/432859**

Filed: **February 05, 2024**

Prior Publication Data

Document Identifier	Publication Date
US 20240184739 A1	Jun. 06, 2024

Related U.S. Application Data

continuation parent-doc US 17310540 US 11954062 WO PCT/US2020/022838 20200314 child-doc US 18432859
us-provisional-application US 62819337 20190315
us-provisional-application US 62819361 20190315
us-provisional-application US 62819435 20190315

Publication Classification

Int. Cl.: **G06F12/00** (20060101); **G06F7/544** (20060101); **G06F7/575** (20060101); **G06F7/58** (20060101); **G06F9/30** (20180101); **G06F9/38** (20180101); **G06F9/50** (20060101); **G06F12/02** (20060101); **G06F12/06** (20060101); **G06F12/0802** (20160101); **G06F12/0804** (20160101); **G06F12/0811** (20160101); **G06F12/0862** (20160101); **G06F12/0866** (20160101); **G06F12/0871** (20160101); **G06F12/0875** (20160101); **G06F12/0882** (20160101); **G06F12/0888** (20160101); **G06F12/0891** (20160101); **G06F12/0893** (20160101); **G06F12/0895** (20160101); **G06F12/0897** (20160101); **G06F12/1009** (20160101); **G06F12/128** (20160101); **G06F15/78** (20060101); **G06F15/80** (20060101); **G06F17/16** (20060101); **G06F17/18** (20060101); **G06T1/20** (20060101); **G06T1/60** (20060101); **H03M7/46** (20060101); G06N3/08 (20230101); G06T15/06 (20110101)

U.S. Cl.:

CPC **G06F15/7839** (20130101); **G06F7/5443** (20130101); **G06F7/575** (20130101); **G06F7/588** (20130101); **G06F9/3001** (20130101); **G06F9/30014** (20130101); **G06F9/30036** (20130101); **G06F9/3004** (20130101); **G06F9/30043** (20130101); **G06F9/30047** (20130101); **G06F9/30065** (20130101); **G06F9/30079** (20130101); **G06F9/3887** (20130101); **G06F9/3888** (20230801); **G06F9/5011** (20130101); **G06F9/5077** (20130101); **G06F12/0215** (20130101); **G06F12/0238** (20130101); **G06F12/0246** (20130101); **G06F12/0607** (20130101); **G06F12/0802** (20130101); **G06F12/0804** (20130101); **G06F12/0811** (20130101); **G06F12/0862** (20130101); **G06F12/0866** (20130101); **G06F12/0871** (20130101); **G06F12/0875** (20130101); **G06F12/0882** (20130101); **G06F12/0888** (20130101); **G06F12/0891** (20130101); **G06F12/0893** (20130101); **G06F12/0895** (20130101); **G06F12/0897** (20130101); **G06F12/1009** (20130101); **G06F12/128** (20130101); **G06F15/8046** (20130101); **G06F17/16** (20130101); **G06F17/18** (20130101); **G06T1/20** (20130101); **G06T1/60** (20130101); **H03M7/46** (20130101); G06F9/3802 (20130101); G06F9/3818 (20130101); G06F9/3867 (20130101); G06F2212/1008 (20130101); G06F2212/1021 (20130101); G06F2212/1044 (20130101); G06F2212/302 (20130101); G06F2212/401 (20130101); G06F2212/455 (20130101); G06F2212/60 (20130101); G06N3/08 (20130101); G06T15/06 (20130101)

Field of Classification Search

CPC: G06F (15/7839); G06F (7/5443); G06F (7/575); G06F (7/588); G06F (9/3001); G06F (9/30014); G06F (9/30036); G06F (9/3004); G06F (9/30043); G06F (9/30047); G06F (9/30065); G06F (9/30079); G06F (9/3887); G06F (9/3888); G06F (9/5011); G06F (9/5077); G06F (12/0215); G06F (12/0238); G06F (12/0246); G06F (12/0607); G06F

(12/0802); G06F (12/0804); G06F (12/0811); G06F (12/0862); G06F (12/0866); G06F (12/0871); G06F (12/0875); G06F (12/0882); G06F (12/0888); G06F (12/0891); G06F (12/0893); G06F (12/0895); G06F (12/0897); G06F (12/1009); G06F (12/128); G06F (15/8046); G06F (17/16); G06F (17/18); G06F (9/3802); G06F (9/3818); G06F (9/3867); G06F (2212/1008); G06F (2212/1021); G06F (2212/1044); G06F (2212/302); G06F (2212/401); G06F (2212/455); G06F (2212/60); G06F (2212/1016); G06F (7/58); G06F (9/5066); G06F (2212/1024); G06F (15/173); G06F (16/24569); G06F (12/12); G06F (2212/2542); G06F (2212/601); G06F (2212/608); G06F (2212/652); G06F (9/383); G06F (2212/6026); G06F (2212/6028); G06F (12/0877); G06T (1/20); G06T (1/60); G06T (15/06); H03M (7/46); G06N (3/08)

References Cited

U.S. PATENT DOCUMENTS

Patent No.	Issued Date	Patentee Name	U.S. Cl.	CPC
3872442	12/1974	Boles et al.	N/A	N/A
4476523	12/1983	Beauchamp	N/A	N/A
4823252	12/1988	Horst et al.	N/A	N/A
5182801	12/1992	Asfour	N/A	N/A
5381539	12/1994	Yanai et al.	N/A	N/A
5450607	12/1994	Kowalczyk et al.	N/A	N/A
5469552	12/1994	Suzuki et al.	N/A	N/A
5471593	12/1994	Branigin	712/E9.05	G06F 9/30018
5502827	12/1995	Yoshida	N/A	N/A
5574928	12/1995	White et al.	N/A	N/A
5627985	12/1996	Fetterman et al.	N/A	N/A
5673407	12/1996	Poland et al.	N/A	N/A
5737752	12/1997	Hilditch	N/A	N/A
5777629	12/1997	Baldwin	N/A	N/A
5805475	12/1997	Putrino et al.	N/A	N/A
5822767	12/1997	Macwillliams et al.	N/A	N/A
5890211	12/1998	Sokolov et al.	N/A	N/A
5917741	12/1998	Ng	N/A	N/A
5926406	12/1998	Tucker et al.	N/A	N/A
5940311	12/1998	Dao et al.	N/A	N/A
5943687	12/1998	Liedberg	N/A	N/A
6038582	12/1999	Arakawa et al.	N/A	N/A
6049865	12/1999	Smith	N/A	N/A
6078940	12/1999	Scales	N/A	N/A
6260008	12/2000	Sanfilippo	N/A	N/A
6412046	12/2001	Sharma et al.	N/A	N/A
6480872	12/2001	Choquette	N/A	N/A
6513099	12/2002	Smith et al.	N/A	N/A
6529928	12/2002	Resnick et al.	N/A	N/A
6578102	12/2002	Batchelor et al.	N/A	N/A
6598120	12/2002	Berg et al.	N/A	N/A
6678806	12/2003	Redford	N/A	N/A
6788738	12/2003	New	N/A	N/A

6856320	12/2004	Rubinstein et al.	N/A	N/A
6947049	12/2004	Spitzer et al.	N/A	N/A
6963954	12/2004	Trehus et al.	N/A	N/A
7102646	12/2005	Rubinstein et al.	N/A	N/A
7127482	12/2005	Hou et al.	N/A	N/A
7197605	12/2006	Schmisseur et al.	N/A	N/A
7327289	12/2007	Lippincott	N/A	N/A
7346741	12/2007	Keish et al.	N/A	N/A
7373369	12/2007	Gerwig et al.	N/A	N/A
7483031	12/2008	Williams et al.	N/A	N/A
7567252	12/2008	Buck et al.	N/A	N/A
7616206	12/2008	Danilak	N/A	N/A
7620793	12/2008	Edmondson et al.	N/A	N/A
7873812	12/2010	Mimar	N/A	N/A
7913041	12/2010	Shen et al.	N/A	N/A
7958558	12/2010	Leake et al.	N/A	N/A
8253750	12/2011	Huang et al.	N/A	N/A
8340280	12/2011	Gueron et al.	N/A	N/A
8429351	12/2012	Yu et al.	N/A	N/A
8488055	12/2012	Côté et al.	N/A	N/A
8645634	12/2013	Cox et al.	N/A	N/A
8669990	12/2013	Sprangle et al.	N/A	N/A
8847965	12/2013	Chandak et al.	N/A	N/A
8990505	12/2014	Jamil et al.	N/A	N/A
9104633	12/2014	Moloney	N/A	N/A
9170955	12/2014	Forsyth et al.	N/A	N/A
9304835	12/2015	Ekanadham et al.	N/A	N/A
9317251	12/2015	Tsen et al.	N/A	N/A
9411468	12/2015	Hooker et al.	N/A	N/A
9461667	12/2015	Quinnell	N/A	N/A
9491112	12/2015	Patel et al.	N/A	N/A
9501392	12/2015	Weingarten	N/A	N/A
9558156	12/2016	Bekas et al.	N/A	N/A
9690696	12/2016	Hefner et al.	N/A	N/A
9727337	12/2016	Gschwind et al.	N/A	N/A
9804666	12/2016	Jiao	N/A	N/A
9811468	12/2016	Hooker et al.	N/A	N/A
9960917	12/2017	Gopal et al.	N/A	N/A
10002045	12/2017	Chung et al.	N/A	N/A
10049322	12/2017	Ross	N/A	N/A
10102015	12/2017	Gordon et al.	N/A	N/A
10146738	12/2017	Nurvitadhi et al.	N/A	N/A
10229470	12/2018	Benthin et al.	N/A	N/A
10353706	12/2018	Kaul et al.	N/A	N/A
10379865	12/2018	Menezes et al.	N/A	N/A
10409614	12/2018	Ould-Ahmed-Vall et al.	N/A	N/A
10409887	12/2018	Gauria et al.	N/A	N/A
10474458	12/2018	Kaul et al.	N/A	N/A
10528864	12/2019	Dally et al.	N/A	N/A

10572409	12/2019	Zejda et al.	N/A	N/A
10678508	12/2019	Vantrease et al.	N/A	N/A
10755201	12/2019	Sika	N/A	N/A
10762137	12/2019	Volpe et al.	N/A	N/A
10762164	12/2019	Chen et al.	N/A	N/A
10769750	12/2019	Yoon	N/A	N/A
10860316	12/2019	Zhi et al.	N/A	N/A
10860922	12/2019	Dally et al.	N/A	N/A
10891538	12/2020	Dally et al.	N/A	N/A
10896045	12/2020	Sodani et al.	N/A	N/A
11080046	12/2020	Kaul et al.	N/A	N/A
11113784	12/2020	Ray et al.	N/A	N/A
11169799	12/2020	Kaul et al.	N/A	N/A
11250108	12/2021	Dong	N/A	N/A
11360767	12/2021	Kaul et al.	N/A	N/A
11361496	12/2021	Maiyuran et al.	N/A	N/A
11409537	12/2021	Ould-Ahmed-Vall et al.	N/A	N/A
11461107	12/2021	Ould-Ahmed-Vall et al.	N/A	N/A
11550971	12/2022	Lu et al.	N/A	N/A
11620256	12/2022	Koker et al.	N/A	N/A
11805109	12/2022	Khare et al.	N/A	N/A
12007935	12/2023	Maiyuran et al.	N/A	N/A
2001/0042194	12/2000	Elliott et al.	N/A	N/A
2002/0152361	12/2001	Dean et al.	N/A	N/A
2002/0156979	12/2001	Rodriguez	N/A	N/A
2002/0188808	12/2001	Rowlands et al.	N/A	N/A
2003/0204840	12/2002	Wu	N/A	N/A
2004/0054841	12/2003	Callison et al.	N/A	N/A
2005/0080834	12/2004	Belluomini et al.	N/A	N/A
2005/0125631	12/2004	Symes et al.	N/A	N/A
2005/0169463	12/2004	Ahn et al.	N/A	N/A
2005/0219253	12/2004	Piazza et al.	N/A	N/A
2006/0037003	12/2005	Long et al.	N/A	N/A
2006/0083489	12/2005	Aridome et al.	N/A	N/A
2006/0101244	12/2005	Siu	N/A	N/A
2006/0143396	12/2005	Cabot	N/A	N/A
2006/0149803	12/2005	Siu et al.	N/A	N/A
2006/0179092	12/2005	Schmookler	N/A	N/A
2006/0248279	12/2005	Al-Sukhni et al.	N/A	N/A
2006/0265576	12/2005	Davis et al.	N/A	N/A
2006/0277365	12/2005	Pong	N/A	N/A
2006/0282620	12/2005	Kashyap et al.	N/A	N/A
2007/0030277	12/2006	Prokopenko et al.	N/A	N/A
2007/0030279	12/2006	Paltashev et al.	N/A	N/A
2007/0074008	12/2006	Donofrio	N/A	N/A
2007/0083742	12/2006	Abernathy et al.	N/A	N/A
2007/0115291	12/2006	Chen et al.	N/A	N/A
2007/0211064	12/2006	Buck et al.	N/A	N/A

2007/0273698	12/2006	Du et al.	N/A	N/A
2007/0294682	12/2006	Demetriou et al.	N/A	N/A
2008/0028152	12/2007	Du et al.	N/A	N/A
2008/0030510	12/2007	Wan et al.	N/A	N/A
2008/0052466	12/2007	Zulauf	N/A	N/A
2008/0071851	12/2007	Zohar et al.	N/A	N/A
2008/0086598	12/2007	Maron et al.	N/A	N/A
2008/0189487	12/2007	Craske	N/A	N/A
2008/0288746	12/2007	Inglett et al.	N/A	N/A
2008/0307207	12/2007	Khailany et al.	N/A	N/A
2009/0003593	12/2008	Gopal et al.	N/A	N/A
2009/0019253	12/2008	Strecher et al.	N/A	N/A
2009/0030960	12/2008	Geraghty et al.	N/A	N/A
2009/0113170	12/2008	Abdallah	N/A	N/A
2009/0150654	12/2008	Oberman et al.	N/A	N/A
2009/0157972	12/2008	Byers et al.	N/A	N/A
2009/0182942	12/2008	Greiner et al.	N/A	N/A
2009/0189898	12/2008	Dammertz	N/A	N/A
2009/0190432	12/2008	Bilger et al.	N/A	N/A
2009/0254733	12/2008	Chen et al.	N/A	N/A
2009/0307472	12/2008	Essick, IV et al.	N/A	N/A
2009/0309896	12/2008	DeLaurier et al.	N/A	N/A
2010/0053162	12/2009	Dammertz	N/A	N/A
2010/0082906	12/2009	Hinton et al.	N/A	N/A
2010/0082910	12/2009	Raikar et al.	N/A	N/A
2010/0162247	12/2009	Welc et al.	N/A	N/A
2010/0185816	12/2009	Sauber et al.	N/A	N/A
2010/0228941	12/2009	Koob et al.	N/A	N/A
2010/0281235	12/2009	Vorbach et al.	N/A	N/A
2010/0293334	12/2009	Xun et al.	N/A	N/A
2010/0299656	12/2009	Shah et al.	N/A	N/A
2010/0312944	12/2009	Walker	N/A	N/A
2010/0332775	12/2009	Kapil et al.	N/A	N/A
2011/0040744	12/2010	Haas	N/A	N/A
2011/0040822	12/2010	Eichenberger et al.	N/A	N/A
2011/0060879	12/2010	Rogers et al.	N/A	N/A
2011/0078226	12/2010	Baskaran et al.	N/A	N/A
2011/0119446	12/2010	Blumrich	N/A	N/A
2011/0153707	12/2010	Ginzburg et al.	N/A	N/A
2011/0157195	12/2010	Sprangle et al.	N/A	N/A
2011/0238934	12/2010	Xu et al.	N/A	N/A
2012/0011182	12/2011	Raafat et al.	N/A	N/A
2012/0059983	12/2011	Nellans et al.	N/A	N/A
2012/0072631	12/2011	Chirca et al.	N/A	N/A
2012/0075319	12/2011	Dally	N/A	N/A
2012/0124115	12/2011	Ollmann	N/A	N/A
2012/0233444	12/2011	Stephens et al.	N/A	N/A
2012/0268909	12/2011	Emma et al.	N/A	N/A
2012/0278376	12/2011	Bakos	N/A	N/A
2013/0013864	12/2012	Chung et al.	N/A	N/A

2013/0031328	12/2012	Kelleher	N/A	N/A
2013/0073599	12/2012	Maloney	N/A	N/A
2013/0099946	12/2012	Dickie	N/A	N/A
2013/0111136	12/2012	Bell, Jr. et al.	N/A	N/A
2013/0141442	12/2012	Brothers et al.	N/A	N/A
2013/0148947	12/2012	Glen et al.	N/A	N/A
2013/0185515	12/2012	Sassone et al.	N/A	N/A
2013/0218938	12/2012	Dockser et al.	N/A	N/A
2013/0219088	12/2012	Rawe et al.	N/A	N/A
2013/0235925	12/2012	Nguyen et al.	N/A	N/A
2013/0254491	12/2012	Coleman et al.	N/A	N/A
2013/0293544	12/2012	Schreyer et al.	N/A	N/A
2013/0297906	12/2012	Loh et al.	N/A	N/A
2014/0006753	12/2013	Gopal et al.	N/A	N/A
2014/0032818	12/2013	Chang et al.	N/A	N/A
2014/0052928	12/2013	Shimoi	N/A	N/A
2014/0059328	12/2013	Gonion	N/A	N/A
2014/0068197	12/2013	Joshi et al.	N/A	N/A
2014/0075060	12/2013	Sharp et al.	N/A	N/A
2014/0075163	12/2013	Loewenstein et al.	N/A	N/A
2014/0082322	12/2013	Loh et al.	N/A	N/A
2014/0089371	12/2013	Dinechin et al.	N/A	N/A
2014/0089697	12/2013	Kim et al.	N/A	N/A
2014/0095796	12/2013	Bell, Jr. et al.	N/A	N/A
2014/0108481	12/2013	Davis et al.	N/A	N/A
2014/0122555	12/2013	Hickmann et al.	N/A	N/A
2014/0129807	12/2013	Tannenbaum et al.	N/A	N/A
2014/0146607	12/2013	Nagai et al.	N/A	N/A
2014/0173203	12/2013	Forsyth	N/A	N/A
2014/0173207	12/2013	Wang et al.	N/A	N/A
2014/0181487	12/2013	Sasanka	N/A	N/A
2014/0188963	12/2013	Tsen et al.	N/A	N/A
2014/0188966	12/2013	Galal et al.	N/A	N/A
2014/0208069	12/2013	Wegener	N/A	N/A
2014/0223096	12/2013	Zhe Yang et al.	N/A	N/A
2014/0223131	12/2013	Agarwal et al.	N/A	N/A
2014/0229713	12/2013	Muff et al.	N/A	N/A
2014/0258622	12/2013	Lacourba et al.	N/A	N/A
2014/0266417	12/2013	Dally et al.	N/A	N/A
2014/0267232	12/2013	Lum	N/A	N/A
2014/0281008	12/2013	Muthiah et al.	N/A	N/A
2014/0281110	12/2013	Duluk, Jr. et al.	N/A	N/A
2014/0281261	12/2013	Vera et al.	N/A	N/A
2014/0281299	12/2013	Duluk, Jr. et al.	N/A	N/A
2014/0317388	12/2013	Chung et al.	N/A	N/A
2014/0348431	12/2013	Brick et al.	N/A	N/A
2014/0365715	12/2013	Lee	N/A	N/A
2014/0379987	12/2013	Aggarwal et al.	N/A	N/A
2015/0039661	12/2014	Blomgren et al.	N/A	N/A
2015/0046655	12/2014	Nystad et al.	N/A	N/A

2015/0046662	12/2014	Heinrich	N/A	N/A
2015/0067259	12/2014	Wang et al.	N/A	N/A
2015/0095588	12/2014	Abdallah	N/A	N/A
2015/0160872	12/2014	Chen	N/A	N/A
2015/0193358	12/2014	Dyke et al.	N/A	N/A
2015/0205615	12/2014	Cunningham	N/A	N/A
2015/0205724	12/2014	Hancock et al.	N/A	N/A
2015/0221063	12/2014	Kim et al.	N/A	N/A
2015/0235338	12/2014	Alla et al.	N/A	N/A
2015/0261683	12/2014	Hong et al.	N/A	N/A
2015/0268940	12/2014	Baghsorkhi et al.	N/A	N/A
2015/0268963	12/2014	Etsion et al.	N/A	N/A
2015/0278984	12/2014	Koker et al.	N/A	N/A
2015/0334043	12/2014	Li et al.	N/A	N/A
2015/0339229	12/2014	Zhang et al.	N/A	N/A
2015/0349953	12/2014	Kruglick	N/A	N/A
2015/0371407	12/2014	Kim	N/A	N/A
2015/0378741	12/2014	Lukyanov et al.	N/A	N/A
2015/0378920	12/2014	Gierach et al.	N/A	N/A
2015/0380088	12/2014	Naeimi et al.	N/A	N/A
2016/0062947	12/2015	Chetlur et al.	N/A	N/A
2016/0070536	12/2015	Maeda	N/A	N/A
2016/0071242	12/2015	Uralsky et al.	N/A	N/A
2016/0086303	12/2015	Bae et al.	N/A	N/A
2016/0092118	12/2015	Kumar et al.	N/A	N/A
2016/0092239	12/2015	Maiyuran et al.	N/A	N/A
2016/0092240	12/2015	Maiyuran et al.	N/A	N/A
2016/0124713	12/2015	Bekas et al.	N/A	N/A
2016/0124861	12/2015	Fujii et al.	N/A	N/A
2016/0140686	12/2015	Lueh et al.	N/A	N/A
2016/0170889	12/2015	Lee et al.	N/A	N/A
2016/0179535	12/2015	Chen et al.	N/A	N/A
2016/0188295	12/2015	Rarick et al.	N/A	N/A
2016/0239065	12/2015	Lee et al.	N/A	N/A
2016/0246723	12/2015	Doshi et al.	N/A	N/A
2016/0255169	12/2015	Kovvuri et al.	N/A	N/A
2016/0283249	12/2015	Forsell	N/A	N/A
2016/0307362	12/2015	Farrell et al.	N/A	N/A
2016/0321187	12/2015	Bernat et al.	N/A	N/A
2016/0342892	12/2015	Ross	N/A	N/A
2016/0350227	12/2015	Hooker et al.	N/A	N/A
2016/0357680	12/2015	Hooker et al.	N/A	N/A
2016/0378366	12/2015	Tomishima et al.	N/A	N/A
2016/0378465	12/2015	Venkatesh et al.	N/A	N/A
2017/0024327	12/2016	Saidi et al.	N/A	N/A
2017/0039144	12/2016	Wang et al.	N/A	N/A
2017/0061279	12/2016	Yang et al.	N/A	N/A
2017/0083240	12/2016	Rogers et al.	N/A	N/A
2017/0090924	12/2016	Mishra et al.	N/A	N/A
2017/0109282	12/2016	Frank et al.	N/A	N/A

2017/0132134	12/2016	Gschwind et al.	N/A	N/A
2017/0139748	12/2016	Tipparaju et al.	N/A	N/A
2017/0177336	12/2016	Astafev et al.	N/A	N/A
2017/0185379	12/2016	Anderson et al.	N/A	N/A
2017/0200303	12/2016	Havran	N/A	N/A
2017/0214930	12/2016	Loughry	N/A	N/A
2017/0220592	12/2016	Foltz	N/A	G06F 12/0806
2017/0228342	12/2016	Narayanaswami et al.	N/A	N/A
2017/0277460	12/2016	Li et al.	N/A	N/A
2017/0308800	12/2016	Cichon et al.	N/A	N/A
2017/0315921	12/2016	Hooker et al.	N/A	N/A
2017/0315932	12/2016	Moyer	N/A	N/A
2017/0323042	12/2016	Zhang	N/A	N/A
2017/0344822	12/2016	Popescu et al.	N/A	N/A
2017/0357506	12/2016	Wang et al.	N/A	N/A
2018/0011790	12/2017	Gaur et al.	N/A	N/A
2018/0018153	12/2017	Mukai	N/A	N/A
2018/0018266	12/2017	Jones, III	N/A	N/A
2018/0026651	12/2017	Gopal	N/A	N/A
2018/0037320	12/2017	Shi et al.	N/A	N/A
2018/0040096	12/2017	Benthin et al.	N/A	N/A
2018/0046898	12/2017	Lo	N/A	N/A
2018/0046900	12/2017	Dally et al.	N/A	N/A
2018/0046906	12/2017	Dally et al.	N/A	N/A
2018/0052920	12/2017	Klein et al.	N/A	N/A
2018/0067869	12/2017	Yang et al.	N/A	N/A
2018/0107602	12/2017	Dasgupta	N/A	N/A
2018/0114114	12/2017	Molchanov et al.	N/A	N/A
2018/0129608	12/2017	Damodaran et al.	N/A	N/A
2018/0157464	12/2017	Lutz et al.	N/A	N/A
2018/0165204	12/2017	Venkatesh et al.	N/A	N/A
2018/0173623	12/2017	Koob et al.	N/A	N/A
2018/0174353	12/2017	Shin	N/A	N/A
2018/0181519	12/2017	Cheng	N/A	N/A
2018/0183577	12/2017	Suresh et al.	N/A	N/A
2018/0189231	12/2017	Fleming, Jr. et al.	N/A	N/A
2018/0189925	12/2017	Lee et al.	N/A	N/A
2018/0210830	12/2017	Malladi et al.	N/A	N/A
2018/0210836	12/2017	Lai et al.	N/A	N/A
2018/0232846	12/2017	Gruber et al.	N/A	N/A
2018/0253635	12/2017	Park	N/A	N/A
2018/0276150	12/2017	Eckert et al.	N/A	N/A
2018/0284994	12/2017	Haller et al.	N/A	N/A
2018/0285261	12/2017	Mandal et al.	N/A	N/A
2018/0285264	12/2017	Kayiran et al.	N/A	N/A
2018/0285278	12/2017	Appu et al.	N/A	N/A
2018/0286053	12/2017	Labbe et al.	N/A	N/A
2018/0293173	12/2017	Zhu et al.	N/A	N/A

2018/0293784	12/2017	Benthin et al.	N/A	N/A
2018/0293965	12/2017	Vembu et al.	N/A	N/A
2018/0295039	12/2017	Yasman et al.	N/A	N/A
2018/0300258	12/2017	Wokhlu et al.	N/A	N/A
2018/0307494	12/2017	Ould-Ahmed-Vall et al.	N/A	N/A
2018/0307498	12/2017	Yang et al.	N/A	N/A
2018/0315159	12/2017	Ould-Ahmed-Vall et al.	N/A	N/A
2018/0315398	12/2017	Kaul et al.	N/A	N/A
2018/0315399	12/2017	Kaul et al.	N/A	N/A
2018/0321938	12/2017	Boswell et al.	N/A	N/A
2018/0322387	12/2017	Sridharan et al.	N/A	N/A
2018/0336136	12/2017	Hijaz et al.	N/A	N/A
2018/0373200	12/2017	Shi et al.	N/A	N/A
2018/0373635	12/2017	Mukherjee et al.	N/A	N/A
2018/0373809	12/2017	Ylitie	N/A	N/A
2019/0035140	12/2018	Fricke et al.	N/A	N/A
2019/0042193	12/2018	Pasca et al.	N/A	N/A
2019/0042237	12/2018	Azizi et al.	N/A	N/A
2019/0042244	12/2018	Gregory et al.	N/A	N/A
2019/0042457	12/2018	Doshi et al.	N/A	N/A
2019/0042534	12/2018	Butera et al.	N/A	N/A
2019/0042542	12/2018	Narayanamoorthy et al.	N/A	N/A
2019/0042923	12/2018	Pasca	N/A	N/A
2019/0065051	12/2018	Mills et al.	N/A	N/A
2019/0065150	12/2018	Heddes et al.	N/A	N/A
2019/0065195	12/2018	Pool et al.	N/A	N/A
2019/0065338	12/2018	Bramley	N/A	N/A
2019/0066255	12/2018	Nalluri et al.	N/A	N/A
2019/0073590	12/2018	Wu et al.	N/A	N/A
2019/0079767	12/2018	Heinecke	N/A	N/A
2019/0079768	12/2018	Heinecke	N/A	N/A
2019/0095223	12/2018	Dubel et al.	N/A	N/A
2019/0095336	12/2018	Barczak	N/A	N/A
2019/0108651	12/2018	Gu et al.	N/A	N/A
2019/0114534	12/2018	Teng	N/A	N/A
2019/0114535	12/2018	Ng et al.	N/A	N/A
2019/0121638	12/2018	Knowles	N/A	N/A
2019/0121679	12/2018	Wilkinson et al.	N/A	N/A
2019/0121680	12/2018	Wilkinson et al.	N/A	N/A
2019/0129847	12/2018	Roh	N/A	N/A
2019/0130271	12/2018	Narang et al.	N/A	N/A
2019/0146800	12/2018	Ould-Ahmed-Vall et al.	N/A	N/A
2019/0155768	12/2018	Wilkinson et al.	N/A	N/A
2019/0164050	12/2018	Chen	N/A	N/A
2019/0164268	12/2018	Gallo	N/A	N/A
2019/0179757	12/2018	Walker et al.	N/A	N/A

2019/0188142	12/2018	Rappoport et al.	N/A	N/A
2019/0205358	12/2018	Diril et al.	N/A	N/A
2019/0221556	12/2018	Gomes et al.	N/A	N/A
2019/0227936	12/2018	Jang	N/A	N/A
2019/0251034	12/2018	Bernat et al.	N/A	N/A
2019/0266217	12/2018	Arakawa et al.	N/A	N/A
2019/0278593	12/2018	Elango et al.	N/A	N/A
2019/0278600	12/2018	Frumkin et al.	N/A	N/A
2019/0294413	12/2018	Vantrease et al.	N/A	N/A
2019/0327124	12/2018	Lai et al.	N/A	N/A
2019/0347043	12/2018	Hasbun et al.	N/A	N/A
2019/0361954	12/2018	Page	N/A	N/A
2019/0369988	12/2018	Kaul et al.	N/A	N/A
2019/0370173	12/2018	Boyer et al.	N/A	N/A
2019/0392287	12/2018	Ovsiannikov et al.	N/A	N/A
2020/0021540	12/2019	Marolia et al.	N/A	N/A
2020/0042362	12/2019	Cui et al.	N/A	N/A
2020/0050830	12/2019	Krueger et al.	N/A	N/A
2020/0058155	12/2019	Bakalash et al.	N/A	N/A
2020/0061811	12/2019	Iqbal	N/A	N/A
2020/0065241	12/2019	Cho et al.	N/A	N/A
2020/0073825	12/2019	Zaydman	N/A	N/A
2020/0081714	12/2019	Britto et al.	N/A	N/A
2020/0097409	12/2019	Nathella et al.	N/A	N/A
2020/0097411	12/2019	Pusdesris et al.	N/A	N/A
2020/0098725	12/2019	Liff et al.	N/A	N/A
2020/0117463	12/2019	Nassi et al.	N/A	N/A
2020/0117999	12/2019	Yoon et al.	N/A	N/A
2020/0133898	12/2019	Therene et al.	N/A	N/A
2020/0150926	12/2019	Connor et al.	N/A	N/A
2020/0174897	12/2019	McNamara et al.	N/A	N/A
2020/0175074	12/2019	Li et al.	N/A	N/A
2020/0184309	12/2019	Patel	N/A	N/A
2020/0201810	12/2019	Felix et al.	N/A	N/A
2020/0202195	12/2019	Patel	N/A	N/A
2020/0203332	12/2019	Schreiber et al.	N/A	N/A
2020/0210192	12/2019	Alexander et al.	N/A	N/A
2020/0211147	12/2019	Doyle et al.	N/A	N/A
2020/0211253	12/2019	Liktor et al.	N/A	N/A
2020/0211267	12/2019	Rowley et al.	N/A	N/A
2020/0218538	12/2019	Mansell	N/A	N/A
2020/0218540	12/2019	Kesiraju et al.	N/A	N/A
2020/0226920	12/2019	Takenaka et al.	N/A	N/A
2020/0234124	12/2019	Park	N/A	N/A
2020/0242049	12/2019	Loh et al.	N/A	N/A
2020/0250098	12/2019	Ma et al.	N/A	N/A
2020/0285592	12/2019	Ambroladze et al.	N/A	N/A
2020/0311531	12/2019	Liu et al.	N/A	N/A
2020/0371752	12/2019	Murray et al.	N/A	N/A
2020/0380369	12/2019	Case et al.	N/A	N/A

2021/0012197	12/2020	Simonyan et al.	N/A	N/A
2021/0019591	12/2020	Venkatesh et al.	N/A	N/A
2021/0034979	12/2020	Robinson et al.	N/A	N/A
2021/0035258	12/2020	Ray et al.	N/A	N/A
2021/0103550	12/2020	Appu et al.	N/A	N/A
2021/0117201	12/2020	Gray	N/A	N/A
2021/0124579	12/2020	Kaul et al.	N/A	N/A
2021/0150663	12/2020	Maiyuran et al.	N/A	N/A
2021/0150770	12/2020	Appu et al.	N/A	N/A
2021/0182024	12/2020	Mueller et al.	N/A	N/A
2021/0182058	12/2020	Kaul et al.	N/A	N/A
2021/0182140	12/2020	Gruber	N/A	N/A
2021/0200680	12/2020	Vogelsang et al.	N/A	N/A
2021/0211643	12/2020	Da Silva	N/A	N/A
2021/0216316	12/2020	Bhoria et al.	N/A	N/A
2021/0224125	12/2020	Liu et al.	N/A	N/A
2021/0248085	12/2020	Bian et al.	N/A	N/A
2021/0312697	12/2020	Maiyuran	N/A	N/A
2021/0374897	12/2020	Ray et al.	N/A	N/A
2022/0019431	12/2021	Kaul et al.	N/A	N/A
2022/0066931	12/2021	Ray et al.	N/A	N/A
2022/0066971	12/2021	Brewer et al.	N/A	N/A
2022/0070115	12/2021	Brewer et al.	N/A	N/A
2022/0100518	12/2021	Tomei et al.	N/A	N/A
2022/0107914	12/2021	Koker et al.	N/A	N/A
2022/0114096	12/2021	Striramassarma et al.	N/A	N/A
2022/0114108	12/2021	Koker et al.	N/A	N/A
2022/0121421	12/2021	Appu et al.	N/A	N/A
2022/0122215	12/2021	Ray et al.	N/A	N/A
2022/0129265	12/2021	Appu et al.	N/A	N/A
2022/0129266	12/2021	Maiyuran et al.	N/A	N/A
2022/0129271	12/2021	Appu et al.	N/A	N/A
2022/0129521	12/2021	Surti et al.	N/A	N/A
2022/0137967	12/2021	Koker et al.	N/A	N/A
2022/0138101	12/2021	Appu et al.	N/A	N/A
2022/0138104	12/2021	Koker et al.	N/A	N/A
2022/0138895	12/2021	Raganathan et al.	N/A	N/A
2022/0156202	12/2021	Koker et al.	N/A	N/A
2022/0171710	12/2021	Koker et al.	N/A	N/A
2022/0179787	12/2021	Koker et al.	N/A	N/A
2022/0180467	12/2021	Koker et al.	N/A	N/A
2022/0197664	12/2021	Patterson et al.	N/A	N/A
2022/0197800	12/2021	Appu et al.	N/A	N/A
2022/0197975	12/2021	Adelman et al.	N/A	N/A
2022/0335563	12/2021	Elzur	N/A	N/A
2022/0350751	12/2021	Koker et al.	N/A	N/A
2022/0350768	12/2021	Brewer et al.	N/A	N/A
2022/0357945	12/2021	Kaul et al.	N/A	N/A
2022/0365901	12/2021	Maiyuran et al.	N/A	N/A

2022/0382555	12/2021	Ould-Ahmed-Vall et al.	N/A	N/A
2023/0014565	12/2022	Ray et al.	N/A	N/A
2023/0046506	12/2022	Kaul et al.	N/A	N/A
2023/0095363	12/2022	Hsu et al.	N/A	N/A
2023/0419585	12/2022	Smith et al.	N/A	N/A
2024/0111609	12/2023	George et al.	N/A	N/A
2024/0243829	12/2023	Wei et al.	N/A	N/A

FOREIGN PATENT DOCUMENTS

Patent No.	**Application Date**	**Country**	**CPC**
1040803	12/1997	CN	G06F 11/1641
101133393	12/2007	CN	G06F 12/1009
101238454	12/2007	CN	N/A
100458738	12/2008	CN	G06F 12/1027
102214160	12/2010	CN	N/A
104011676	12/2013	CN	N/A
104603748	12/2014	CN	N/A
105468542	12/2015	CN	N/A
106683036	12/2016	CN	N/A
107688745	12/2017	CN	G06F 12/0882
108268422	12/2017	CN	N/A
105468542	12/2018	CN	N/A
112506567	12/2020	CN	N/A
113396401	12/2020	CN	N/A
111666066	12/2020	CN	N/A
115756384	12/2022	CN	N/A
0656592	12/1994	EP	N/A
2849410	12/2014	EP	N/A
2937794	12/2014	EP	N/A
2849410	12/2017	EP	N/A
3382533	12/2017	EP	N/A
3396533	12/2017	EP	N/A
3407183	12/2017	EP	N/A
3407183	12/2018	EP	N/A
3938915	12/2021	EP	N/A
4270201	12/2022	EP	N/A
2286909	12/1994	GB	N/A
2296155	12/1995	GB	N/A
2296155	12/1995	GB	N/A
2447428	12/2007	GB	N/A
2455401	12/2008	GB	N/A
2612167	12/2022	GB	N/A
H1185969	12/1997	JP	N/A
2011530234	12/2010	JP	N/A

2013246598	12/2012	JP	N/A
2018120589	12/2017	JP	N/A
2019036298	12/2018	JP	N/A
2019029023	12/2019	JP	N/A
2022-523909	12/2021	JP	N/A
2024138299	12/2023	JP	N/A
20120113777	12/2011	KR	N/A
1020120113777	12/2011	KR	N/A
20170052432	12/2016	KR	N/A
1020170093697	12/2016	KR	N/A
20170126999	12/2016	KR	N/A
1020170126999	12/2016	KR	N/A
1020190003849	12/2018	KR	N/A
20190027367	12/2018	KR	N/A
1020170126999	12/2019	KR	N/A
1020190027367	12/2022	KR	N/A
200949691	12/2008	TW	N/A
201344564	12/2012	TW	N/A
201614997	12/2015	TW	N/A
2013095619	12/2012	WO	N/A
2013101018	12/2012	WO	N/A
2013119226	12/2012	WO	N/A
2016097813	12/2015	WO	N/A
2018125250	12/2017	WO	N/A
2018213636	12/2017	WO	N/A
2020/190805	12/2019	WO	N/A
2020190796	12/2019	WO	N/A
2020190797	12/2019	WO	N/A
2020190798	12/2019	WO	N/A
2020190799	12/2019	WO	N/A
2020190800	12/2019	WO	N/A
2020190801	12/2019	WO	N/A
2020190802	12/2019	WO	N/A
2020190803	12/2019	WO	N/A
2020190806	12/2019	WO	N/A
2020190807	12/2019	WO	N/A
2020190808	12/2019	WO	N/A
2020190809	12/2019	WO	N/A
2020190810	12/2019	WO	N/A
2020190811	12/2019	WO	N/A
2020190812	12/2019	WO	N/A
2020190813	12/2019	WO	N/A
2020190814	12/2019	WO	N/A
20201990797	12/2019	WO	N/A
2023069384	12/2022	WO	N/A
2023249742	12/2022	WO	N/A

OTHER PUBLICATIONS

Communication pursuant to Article 94(3) EPC for EP Application 20718900.2, mailed Apr. 10, 2024, 6 pages. cited by applicant

Final OA for U.S. Appl. No. 17/115,989, mailed Mar. 25, 2024, 20 pages. cited by applicant
Final Office Action U.S. Appl. No. 17/430,963 mailed Apr. 19, 2024, 12 pages. cited by applicant
Intent to Grant for EP Application No. 22210195.8, mailed Mar. 28, 2024, 5 pages. cited by applicant
Intention to Grant EP 21192702.5, mailed Apr. 2, 2024, 5 pages. cited by applicant
Notice for Eligibility of Grant for SG Application No. 11202107290Q, 4 pages, Mar. 5, 2024. cited by applicant
Notice of Allowance for JP Application No. 2021-547288, 4 pages, Apr. 25, 2024. cited by applicant
Notice of Allowance for TW 109139554, Mar. 11, 2024, 3 pages. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/590,362 mailed Apr. 3, 2024, 12 pages. cited by applicant
Notification of Issued Patent for TW Application No. 112124508 mailed Mar. 5, 2024, 54 pages. cited by applicant
Notification of Publication of CN202311777921.4, 3 pages, Mar. 12, 2024. cited by applicant
Office Action for CN202110250102.9, mailed Apr. 16, 2024, 13 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,233 mailed Apr. 11, 2024, 16 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,530, mailed Mar. 19, 2024, 8 pages. cited by applicant
Office Action for U.S. Appl. No. 17/961,833, mailed Apr. 18, 2024, 10 pages. cited by applicant
Publication Notification for CN202311810112.9, Mar. 26, 2024, 3 pages. cited by applicant
Zhang et al., “Fabrication cost analysis for 2D, 2.5D, and 3D IC designs,” 2011 IEEE International 3D Systems Integration Conference (3DIC), 2011 IEEE International, Osaka, Japan, 2012, pp. 1-4, doi: 10.1109/3DIC.2012.6263032. cited by applicant
Tawain IPO Search Report for TW Application No. 111130374 mailed Mar. 9, 2023 1 page. cited by applicant
U.S. Appl. No. 17/095,544 “Notice of Allowance” mailed Jan. 31, 2023, 10 pages. cited by applicant
Brunie, “Mixed-precision Fused Multiply and Add”, https://ens-lyon.hal.science/ensl-00642157, Nov. 17, 2011, 7 pages. cited by applicant
Communication pursuant to Article 94(3) for EP20718907.7, mailed Feb. 5, 2024, 6 pages. cited by applicant
Decision to Grant for CN Application No. 202110224132.2, mailed Jan. 5, 2024, 7 pages. cited by applicant
Decision to Grant for JP Application No. 2022-104265, Dec. 5, 2023, 2 pages. cited by applicant
Extended European Search Report for EP Application No. 23197619.2, Jan. 5, 2024, 12 pages. cited by applicant
Final OA for U.S. Appl. No. 17/590,362, Dec. 4, 2023, 30 pages. cited by applicant
Final Office Action for U.S. Appl. No. 17/674,703, mailed Dec. 13, 2023, 26 pages. cited by applicant
Goodfellow et al., “Adaptive Computation and Machine Learning Series,” Book, Nov. 18, 2016, pp. 98-165, Chapter 5, The MIT Press, Cambridge, MA, USA. cited by applicant
International Patent Application No. PCT/US20/22833 “International Preliminary Report on Patentability” issued Sep. 16, 2021, 6 pages. cited by applicant
International Patent Application No. PCT/US20/22833 “International Search Report and Written Opinion” mailed Jul. 8, 2020, 8 pages. cited by applicant
International Patent Application No. PCT/US20/22835 “International Preliminary Report on Patentability” issued Sep. 16, 2021, 7 pages. cited by applicant
International Patent Application No. PCT/US20/22835 “International Search Report and Written Opinion” mailed Jun. 10, 2020, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22836 “International Preliminary Report on

Patentability" issued Sep. 16, 2021, 8 pages. cited by applicant
International Patent Application No. PCT/US20/22836 "International Search Report and Written Opinion" mailed Jun. 26, 2020, 11 pages. cited by applicant
International Patent Application No. PCT/US20/22837 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 9 pages. cited by applicant
International Patent Application No. PCT/US20/22837 "International Search Report and Written Opinion" mailed Sep. 15, 2020, 14 pages. cited by applicant
International Patent Application No. PCT/US20/22838 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 8 pages. cited by applicant
International Patent Application No. PCT/US20/22838 "International Search Report and Written Opinion" mailed Jul. 6, 2020, 11 pages. cited by applicant
International Patent Application No. PCT/US20/22839 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 6 pages. cited by applicant
International Patent Application No. PCT/US20/22839 "International Search Report and Written Opinion" mailed Jul. 8, 2020, 8 pages. cited by applicant
International Patent Application No. PCT/US20/22843 "International Search Report and Written Opinion" mailed Jul. 28, 2020, 13 pages. cited by applicant
International Patent Application No. PCT/US20/22845 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22845 "International Search Report and Written Opinion" mailed Jun. 30, 2020, 13 pages. cited by applicant
Balkesen et al., "Rapid: In-Memory Analytical Query Processing Engine with Extreme Performance per Watt," SIGMOD'18, Jun. 10-15, 2018, Industry 4: Graph databases & Query Processing on Modern Hardware, pp. 1407-1419. (Year: 2018). cited by applicant
Grant Notification for CN Application No. 202110256528.5, May 6, 2024, 9 pages. cited by applicant
Grant Notification for CN Application No. 2021109069843X, mailed Apr. 10, 2024, 9 pages. cited by applicant
Heo et al., "BOSS: Bandwidth-Optimized Search Accelerator for Storage-Class Memory," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 279-291, (Year: 2021). cited by applicant
Intention to Grant EP Application No. 20718903.6, May 7, 2024, 5 pages. cited by applicant
JP Application No. 2021-544279 Decision to Grant and Documents Citing Families of 2021-544279 mailed May 29, 2024. cited by applicant
Notice of Allowance in U.S. Appl. No. 17/431,034 mailed May 16, 2024, 10 pages. cited by applicant
Office Action for U.S. Appl. No. 18/312,079, mailed Jun. 3, 2024, 9 pages. cited by applicant
Publication of TW Application No. 113103959 (Pub No. 202420066), May 16, 2024, 3 pages. cited by applicant
Final Office Action in U.S. Appl. No. 17/430,041 mailed Jan. 6, 2025, 21 pages. cited by applicant
Japanese Application No. 2023-0223711 Decision to Grant mailed Jan. 7, 2025. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/430,963, mailed Oct. 23, 2024, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22851 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 7 pages. cited by applicant
International Patent Application No. PCT/US20/22851 "International Search Report and Written Opinion" mailed Jun. 29, 2020, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22852 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 6 pages. cited by applicant
International Patent Application No. PCT/US20/22852 "International Search Report and Written

Opinion” mailed Jul. 21, 2020, 9 pages. cited by applicant
Nicholas Wilt, “The CUDA Handbook: A Comprehensive Guide to GPU Programming,” Book, Jun. 22, 2013, pp. 41-57, Addison-Wesley Professional, Boston, MA, USA. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/310,540, mailed Nov. 21, 2023, 8 pages. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/428,527, mailed Jan. 26, 2024, 10 pages. cited by applicant
Notice of Allowance issued in U.S. Appl. No. 17/429,873, mailed Jan. 11, 2024, 16 pages. cited by applicant
Notification of Grant of Taiwan Patent Application No. 107105949 mailed Apr. 14, 2022, 43 pages. cited by applicant
Nvidia: “Nvidia Tesla P100 Nvidia Tesla P100 Whitepaper”, Jun. 21, 2016, pp. 1-45, downloaded frorn https://images.nvidia.corn/content/pdf/tesla/whitepaper/pascal-architecture-white paper.pdf. cited by applicant
Nvidia's Next Generation CUDA Compute Architecture: Fermi. Datasheet [online]. Nvidia, 2009 [retrieved on Oct. 3, 2019]. Retrieved from the internet. cited by applicant
Office Action and Search Report for Taiwan Patent Application No. 108141986 mailed Mar. 4, 2022, 24 pages (including translation). cited by applicant
Office Action for U.S. Appl. No. 17/430,611 mailed Jun. 2, 2023, 10 pages. cited by applicant
Office Action for CN 202110214543.3, 12 pages, no translation available. cited by applicant
Office Action for CN202110906984, 6 pages, Nov. 21, 2023. cited by applicant
Office Action for JP 2021-547450, mailed Jun. 6, 2023, 15 pages. cited by applicant
Office Action for TW Application No. 112124508, mailed Aug. 22, 2023, 4 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,216, mailed Jan. 1, 2024, 29 pages. cited by applicant
Office Action for U.S. Appl. No. 17/095,590, mailed Jul. 28, 2023, 23 pages. cited by applicant
Ogg, et al., “Improved Data Compression for Serial Interconnected Network on Chip through Unused Significant Bit Removal”, VLSO Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design. 19th International Conference on, Piscataway, NJ, USA, IEEE Jan. 3, 2006, pp. 525-529, XP010883136. ISBN: 978-0-7695-2502-0. cited by applicant
Ross et al., “Intel Processor Graphics: Architecture & Programming,” Power Point Presentation, Aug. 2015, 78 pages, Intel Corporation, Santa Clara, CA, USA. cited by applicant
Non-Final Office Action mailed Jun. 7, 2024, 14 pages. cited by applicant
Article 94(3) Communication for EP 23181292.6 mailed Jun. 25, 2024, 5 pages. cited by applicant
Korean Office Action for KR2021-7025888, mailed Jul. 29, 2024, 7 pages, No translation available at this time. cited by applicant
Notice of Allowance for CN202011533036.8, 8 pages, Jun. 24, 2024. cited by applicant
Notice of Allowance for CN202110250102.9, Jun. 20, 2024, 7 pages. cited by applicant
Notice of Publication for CN202410567634.9, mailed Aug. 1, 2024, 3 pages. cited by applicant
Notification of Publication for TW113103959, mailed May 20, 2024, 3 pages. cited by applicant
Decision to Grant EP Application No. 20718903.6, mailed Sep. 12, 24, 3 pages. cited by applicant
Non-Final Office Action issued in U.S. Appl. No. 17/430,611, mailed Sep. 11, 2024, 11 pages. cited by applicant
Notice of Allowance and Notice of References Cited issued in U.S. Appl. No. 18/415,052, mailed Sep. 6, 2024, 9 pages. cited by applicant
Notice of Allowance for JP Application No. 2020-189645, dispatched Sep. 3, 2024, 4 pages. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/428,529, mailed Aug. 28, 2024, 9 pages. cited by applicant

Notice of Allowance for U.S. Appl. No. 18/532,245, mailed Sep. 17, 2024, 8 pages. cited by applicant
Office Action for U.S. Appl. No. 18/491,474, mailed Aug. 21, 2024, 7 pages. cited by applicant
Office Action from KR Application No. 2021-07025864, mailed Aug. 20, 2024, 20 pages. cited by applicant
Notice of Reasons for Refusal issued in JP Application No. 2021-544279, mailed Feb. 27, 2024, 7 pages including translation. cited by applicant
Office Action for U.S. Appl. No. 17/862,739, mailed Feb. 28, 2024, 21 pages. cited by applicant
U.S. Appl. No. 17/303,654 "Non-Final Office Action" mailed Oct. 31, 2022, 15 pages. cited by applicant
U.S. Appl. No. 17/304,092 "Non-Final Office Action" mailed Oct. 28, 2021, 17 pages. cited by applicant
U.S. Appl. No. 17/428,539 "Non-Final Office Action" mailed Jan. 5, 2023, 14 pages. cited by applicant
U.S. Appl. No. 17/429,873 "Non-Final Office Action" mailed Jan. 9, 2023, 13 pages. cited by applicant
U.S. Appl. No. 17/430,574 "Non-Final Office Action" mailed Jan. 23, 2023, 8 pages. cited by applicant
U.S. Appl. No. 17/732,308 "Non-Final Office Action" mailed Jul. 27, 2022, 10 pages. cited by applicant
Yin Jieming, et al., "Modular Routing Design for Chiplet-Based Systems", 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), IEEE, Jun. 1, 2018, pp. 726-738, XP033375532. cited by applicant
Communication pursuant to Article 94(3) for EP22210195, mailed Oct. 4, 2023, 7 pages. cited by applicant
Final Office Action for U.S. Appl. No. 17/428,529, mailed Oct. 20, 2023, 20 pages. cited by applicant
Final Office Action for U.S. Appl. No. 17/430,611 mailed Oct. 26, 2023, 11 pages. cited by applicant
Hong, et al., "Attache: Towards Ideal Memory Compression by Mitigating Metadata Bandwidth Overheads.", 2018 (Year: 2018). cited by applicant
Notice of Allowance for U.S. Appl. No. 17/849,201, mailed Oct. 2, 2023, 9 pages. cited by applicant
Notice of Reasons for Refusal in JP Application No. 2021-544544, Mailed Oct. 31, 2023, 4 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,530, mailed Nov. 14, 2023, 7 pages. cited by applicant
U.S. Appl. No. 17/429,277 "Notice of Allowance" mailed Oct. 31, 2023, 11 pages. cited by applicant
U.S. Appl. No. 17/429,291, Final Office Action, mailed Sep. 28, 2023. cited by applicant
U.S. Appl. No. 17/430,963 Non-Final OA mailed Sep. 28, 2023, 18 pages. cited by applicant
Alfredo Buttari et al. "Using Mixed Precision for Sparse Matrix Computations to Enhance the Performance while Achieving 64-bit Accuracy" University of Tennessee Knoxville, Oak Ridge National Laboratory (Year: 2006). cited by applicant
AMD, "Graphics Core Next Architecture, Generation 3", Aug. 2016. (Year: 2016). cited by applicant
Cui Xuewen et al: Directive-Based Partitioning and Pipelining for Graphics Processing Units11 , 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 29, 2017 (May 29, 2017), pp. 575-584, XP033113969. cited by applicant
Decision on Allowance for Taiwan Application No. 110127153, 1 page, mailed Oct. 27, 2022. cited by applicant

International Patent Application No. PCT/US20/022840 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 9 pages. cited by applicant
International Patent Application No. PCT/US20/22840 "International Search Report and Written Opinion" mailed Jul. 3, 2020, 11 pages. cited by applicant
International Patent Application No. PCT/US20/22841 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 7 pages. cited by applicant
International Patent Application No. PCT/US20/22841 "International Search Report and Written Opinion" mailed Jun. 7, 2020, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22843 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 7 pages. cited by applicant
International Patent Application No. PCT/US20/22844 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 10 pages. cited by applicant
International Patent Application No. PCT/US20/22844 "International Search Report and Written Opinion" mailed Jul. 2, 2020, 13 pages. cited by applicant
International Patent Application No. PCT/US20/22846 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 12 pages. cited by applicant
International Patent Application No. PCT/US20/22846 "International Search Report and Written Opinion" mailed Aug. 31, 2020, 16 pages. cited by applicant
International Patent Application No. PCT/US20/22847 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 6 pages. cited by applicant
International Patent Application No. PCT/US20/22847 "International Search Report and Written Opinion" mailed Jul. 6, 2020, 8 pages. cited by applicant
International Patent Application No. PCT/US20/22848 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 9 pages. cited by applicant
International Patent Application No. PCT/US20/22848 "International Search Report and Written Opinion" mailed Jul. 3, 2020, 12 pages. cited by applicant
International Patent Application No. PCT/US20/22849 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 6 pages. cited by applicant
International Patent Application No. PCT/US20/22849 "International Search Report and Written Opinion" mailed Jul. 8, 2020, 9 pages. cited by applicant
International Patent Application No. PCT/US20/22850 "International Preliminary Report on Patentability" issued Sep. 16, 2021, 7 pages. cited by applicant
International Patent Application No. PCT/US20/22850 "International Search Report and Written Opinion" mailed Jul. 21, 2020, 9 pages. cited by applicant
Janzen Johan et al: "Partitioning GPUs for Improved Scalability", 2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), IEEE, Oct. 26, 2016 (Oct. 26, 2016), pp. 42-49, XP033028008. cited by applicant
Kaseridis Dimitris, et al., "Minimalist open-page: A DRAM page-mode scheduling polidy for the many-core era", 2011 44th Annual IEEE/ACM International Symposium on Michoarechtecture ACM, Dec. 3, 2011, p. 24-35, XP033064864. cited by applicant
Luo Cheng et al, "An Efficient Task Partitioning and Scheduling Method for Symmetric Multiple GPU Architecture", 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, IEEE, Jul. 16, 2013 (Jul. 16, 2013), p. 1133-1142, XP032529503. cited by applicant
Mark Harris: "Mixed-Precision Programming 1-5 with CUDA 8", Oct. 19, 2016, XP055509917, 11 pages. cited by applicant
Meng, J. et al. Dynamic wrap subdivision for integrated branch and memory divergence tolerance. ACM SIGARCH Computer Architecture News, Jun. 2010 [online], [retrieved on Feb. 16, 2021]. Retrieved from the internet. cited by applicant
U.S. Appl. No. 15/494,773 "Final Office Action" mailed Oct. 12, 2018, 18 pages. cited by applicant

U.S. Appl. No. 15/494,773 "Non-Final Office Action" mailed Apr. 18, 2018, 14 pages. cited by applicant
U.S. Appl. No. 15/494,773 "Notice of Allowability" mailed Jul. 31, 2019, 2 pages. cited by applicant
U.S. Appl. No. 15/494,773 "Notice of Allowance" mailed Jun. 19, 2019, 8 pages. cited by applicant
U.S. Appl. No. 15/787,129 "Non-Final Office Action" mailed Aug. 27, 2018, 9 pages. cited by applicant
U.S. Appl. No. 15/787,129 "Non-Final Office Action" mailed Mar. 7, 2019, 10 pages. cited by applicant
U.S. Appl. No. 15/787,129 "Notice of Allowance" mailed Jul. 10, 2019, 8 pages. cited by applicant
U.S. Appl. No. 15/819,152 "Final Office Action" mailed Nov. 19, 2018, 9 pages. cited by applicant
U.S. Appl. No. 15/819,152 "Non-Final Office Action" mailed Feb. 28, 2018, 7 pages. cited by applicant
U.S. Appl. No. 15/819,152 "Notice of Allowability" mailed Apr. 11, 2019, 2 pages. cited by applicant
U.S. Appl. No. 15/819,152 "Notice of Allowance" mailed Mar. 19, 2019, 5 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Final Office Action" mailed Aug. 19, 2020, 22 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Final Office Action" mailed Jun. 2, 2021, 20 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Final Office Action" mailed Oct. 1, 2018, 17 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Final Office Action" mailed Oct. 18, 2019, 17 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Non-Final Office Action" mailed Dec. 14, 2021, 20 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Non-Final Office Action" mailed Jan. 26, 2021, 19 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Non-Final Office Action" mailed Mar. 1, 2018, 12 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Non-Final Office Action" mailed Mar. 31, 2020, 19 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Non-Final Office Action" mailed May 2, 2019, 15 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Notice of Allowability" mailed Jul. 7, 2022, 4 pages. cited by applicant
U.S. Appl. No. 15/819,167 "Notice of Allowance" mailed Apr. 12, 2022, 7 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Final Office Action" mailed Jul. 9, 2021, 23 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Final Office Action" mailed Sep. 8, 2020, 20 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Non-Final Office Action" mailed Apr. 8, 2020, 19 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Non-Final Office Action" mailed Dec. 17, 2021, 23 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Non-Final Office Action" mailed Feb. 22, 2021, 23 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Notice of Allowability" mailed Aug. 4, 2022, 2 pages. cited by applicant
U.S. Appl. No. 16/227,645 "Notice of Allowance" mailed May 25, 2022, 7 pages. cited by applicant
U.S. Appl. No. 16/432,402 "Final Office Action" mailed Feb. 19, 2021, 9 pages. cited by applicant
U.S. Appl. No. 16/432,402 "Non-Final Office Action" mailed Jul. 10, 2020, 17 pages. cited by applicant

U.S. Appl. No. 16/432,402 "Notice of Allowance" mailed Jun. 23, 2021, 5 pages. cited by applicant
U.S. Appl. No. 17/064,427 "Final Office Action" mailed Jan. 21, 2021, 13 pages. cited by applicant
U.S. Appl. No. 17/064,427 "Non-Final Office Action" mailed Dec. 7, 2020, 11 pages. cited by applicant
U.S. Appl. No. 17/064,427 "Notice of Allowability" mailed Jun. 10, 2021, 2 pages. cited by applicant
U.S. Appl. No. 17/064,427 "Notice of Allowance" mailed May 12, 2021, 7 pages. cited by applicant
U.S. Appl. No. 17/095,544 "Notice of Allowability" mailed Feb. 10, 2023, 2 pages. cited by applicant
U.S. Appl. No. 17/095,590 "Restriction Requirement" mailed Feb. 9, 2023, 5 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Final Office Action" mailed Apr. 1, 2022, 29 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Final Office Action" mailed Dec. 27, 2022, 23 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Final Office Action" mailed May 11, 2021, 23 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Non-Final Office Action" mailed Aug. 30, 2022, 26 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Non-Final Office Action" mailed Feb. 17, 2021, 20 pages. cited by applicant
U.S. Appl. No. 17/122,905 "Non-Final Office Action" mailed Nov. 16, 2021, 23 pages. cited by applicant
U.S. Appl. No. 17/169,232 "Notice of Allowability" mailed Jun. 14, 2021, 3 pages. cited by applicant
U.S. Appl. No. 17/169,232 "Notice of Allowance" mailed Apr. 7, 2021, 8 pages. cited by applicant
U.S. Appl. No. 17/303,654 "Notice of Allowability" mailed Feb. 27, 2023, 2 pages. cited by applicant
U.S. Appl. No. 17/303,654 "Notice of Allowance" mailed Feb. 14, 2023, 8 pages. cited by applicant
U.S. Appl. No. 17/304,092 "Notice of Allowability" mailed Apr. 20, 2022, 3 pages. cited by applicant
U.S. Appl. No. 17/304,092 "Notice of Allowability" mailed Feb. 28, 2022, 2 pages. cited by applicant
U.S. Appl. No. 17/304,092 "Notice of Allowance" mailed Feb. 11, 2022, 9 pages. cited by applicant
U.S. Appl. No. 17/305,355 "Non-Final Office Action" mailed Nov. 5, 2021, 28 pages. cited by applicant
U.S. Appl. No. 17/305,355 "Notice of Allowability" mailed May 4, 2022, 2 pages. cited by applicant
U.S. Appl. No. 17/305,355 "Notice of Allowance" mailed Feb. 23, 2022, 7 pages. cited by applicant
U.S. Appl. No. 17/428,527 "Non-Final Office Action" mailed Mar. 30, 2023, 20 pages. cited by applicant
U.S. Appl. No. 17/732,308 "Notice of Allowability" mailed Mar. 3, 2023, 2 pages. cited by applicant
U.S. Appl. No. 17/732,308 "Notice of Allowance" mailed Dec. 6, 2022, 5 pages. cited by applicant
U.S. Appl. No. 17/827,067 "Non-Final Office Action" mailed Nov. 25, 2022, 25 pages. cited by applicant
U.S. Appl. No. 17/827,067 "Notice of Allowance" mailed Mar. 2, 2023, 8 pages. cited by applicant

U.S. Appl. No. 17/834,482 “Notice of Allowance” mailed Mar. 8, 2023, 7 pages. cited by applicant
U.S. Appl. No. 17/839,856 “Non-Final Office Action” mailed Feb. 10, 2023, 15 pages. cited by applicant
Office Action for U.S. Appl. No. 17/430,574, mailed Jan. 23, 2023, 8 pages. cited by applicant
Yin et al., “Modular Routing Design for Chiplet-Based Systems”, 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Abstract only. 2 pages. cited by applicant
Ogg et al., “Improved Data Compression for Serial Interconnected Network on Chip through Unused Significant Bit Removal”, 19th International Conference on VLSI Design held jointly with 5th International Conference on Embedded Systems Design (VLSID'06), Abstract only. 2 pages. cited by applicant
Office Action for U.S. Appl. No. 17/429,873, Jan. 9, 2023, 13 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,539, Jan. 5, 2023, 14 pages. cited by applicant
International Search Report and Written Opinion of the International Search Authority for PCT Application No. PCT/US2020/022843 mailed Jul. 28, 2020, 15 pages. cited by applicant
Office Action for U.S. Appl. No. 17/303,654, Oct. 31, 2022, 15 pages. cited by applicant
Office Action for U.S. Appl. No. 17/304,092, mailed Oct. 28, 2021, 17 pages. cited by applicant
International Search Report and Written Opinion from PCT/US2020/022852, 11 pages, Jul. 21, 2020. cited by applicant
Anonymous: “vfloat16 floating-point format”, XP055698197, Aug. 24, 2018, 5 pages, downloaded from https://en.wikipedia.org/wiki/Bfloat16_floating-point_format on Jun. 14, 2021. cited by applicant
Notice of Allowance for U.S. Appl. No. 17/095,544, mailed Jan. 31, 2023, 10 pages. cited by applicant
Communication pursuant to Article 94(3) for EP 22 198 615.1, received on Apr. 14, 2023, 5 pages. cited by applicant
Non-Final OA Issued in U.S. Appl. No. 17/429,277 mailed Jul. 7, 2023, 9 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,216, mailed Jul. 7, 2023, 21 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,523, mailed Aug. 3, 2023, 12 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,534, mailed Jul. 17, 2023, 20 pages. cited by applicant
Office Action for U.S. Appl. No. 17/849,201, mailed Jun. 1, 2023, 6 pages. cited by applicant
Office Action for U.S. Appl. No. 17/428,529, mailed Jun. 7, 2023, 18 pages. cited by applicant
U.S. Appl. No. 17/674,703 Non-Final Office Action mailed Jul. 14, 2023, 23 pages. cited by applicant
U.S. Appl. No. 17/430,611 “Non-Final Office Action” mailed Jun. 2, 2023, 11 pages. cited by applicant
Cano, Alberto: A survey on graphic processing unit computing for large-scale data mining. In: WIREs Data Mining and Knowledge Discovery, 8, Dec. 14, 2017, (Online), 1, S. e1232:1-e1232:24. ISSN 1942-4795. cited by applicant
Non-Final Office Action in U.S. Appl. No. 17/429,291, mailed May 22, 2023, 20 pages. cited by applicant
Office Action for U.S. Appl. No. 17/122,905, mailed Apr. 25, 2023, 24 pages. cited by applicant
Office Action from DE Application No. 112020003700.2, mailed Mar. 16, 2023, 18 pages (no translation included). cited by applicant
Office Action, “Translation of Notice” for TW Application No. 111130374, mailed Mar. 9, 2023, 5 pages. cited by applicant
U.S. Appl. No. 17/430,574 Final Office Action mailed May 24, 2023, 9 pages. cited by applicant
U.S. Appl. No. 17/429,873 “Final Office Action” mailed Apr. 26, 2023, 14 pages. cited by applicant
Bruintjes, T., “Sabrewing: A lightweight architecture for combined floating-point and integer arithmetic.” ACM Transactions on Architecture and Code Optimization (TACO) 8.4(12): 1-22,

2012. cited by applicant
Herrera, A., “Nvidia Grid: Graphics Accelerated VDI with the Visual Performance of a Workstation”, 18 pages, May 2014. cited by applicant
Macri, J., “AMD's Next Generation GPU and High Bandwidth Memory Architecture: FURY”, 26 pages, Aug. 2015. cited by applicant
Office Action for U.S. Appl. No. 17/862,739 mailed Sep. 21, 2023, 19 pages. cited by applicant
Sze, V. et al., “Efficient Processing of Deep Neural Networks: A Tutorial and Survey”, 31 pages, Mar. 27, 2017. cited by applicant
U.S. Appl. No. 17/429,873 “Notice of Allowance” mailed Sep. 7, 2023, 8 pages. cited by applicant
U.S. Appl. No. 17/590,362 “Non-Final Office Action” mailed Jul. 10, 2023, 29 pages. cited by applicant
U.S. Appl. No. 17/967,283 “Non-Final Office Action” mailed Aug. 18, 2023, 23 pages. cited by applicant
Yuan, et al., “Complexity Effective Memory Access Scheduling for Many-Core Accelerator Architectures”, 11 pages, Dec. 2009. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022839, Jul. 8, 2020, 10 pages. cited by applicant
Eriko Nurvitadhi et al: “Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?” Proceedings of the 2017 ACM/SIGDA International Symposium on 1Field-Programmable Gate Arrays, FPGA 17, Feb. 22, 2017, pp. 5-14, XP055542383. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022845, Jun. 30, 2020, 15 pages. cited by applicant
International Search Report and Written Opinion for PCT Application No. PCT/US2020/022838, 11 pages, Jun. 22, 2020. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022852, 11 pages, Jul. 10, 2020. cited by applicant
Anonymous: “bfloat16 floating-point format”, retrieved from wikipedia on May 26, 2020, Aug. 24, 2018, 5 pages. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022836, 13 pages, Jun. 26, 2020. cited by applicant
Young Vinson et al: “Combining HW/SW Mechanisms to Improve NUMA Performance of Multi-GPU Systems”, 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (Micro), IEEE, Oct. 20, 2018 (Oct. 20, 2018), pp. 339-351, XP033473308. cited by applicant
Pearson Carl et al: “NUMA-Aware Data-Transfer Measurements for Power/NVLink Multi-GPU Systems”, Jan. 25, 2019 (Jan. 25, 2019), Robocup 2008: Robot Soccer World Cup XII; pp. 448-454, XP047501461, ISBN: 978-3-319-10403-4 [retrieved on Jan. 25, 2019] p. 452, line 14-p. 453, line 6. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022833, 10 pages, Jul. 8, 2020. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022837, 17 pages, Sep. 15, 2020. cited by applicant
Temam O et al: “Software assistance for data caches”, High-Performance Computer Architecture, 1995. Proceedings., First IEEE Symposium on Raleigh, NC, USA Jan. 22-25, 1995, Los Alamitos, CA, USA.IEEE Comput. Soc, Jan. 22, 1995 (Jan. 22, 1995), pp. 154-163, XP010130123. cited by applicant
Tong Chen et al: “Prefetching irregular references for software cache on cell”, Sixth Annual IEEE/ACM International Symposium on Code Generation and 1Optimization (CGO 08). Boston, MA, USA, Apr. 5-9, 2008, New York, NY: ACM, 2 Penn Plaza, Suite 701 New York NY 10121-0701 USA, Apr. 6, 2008 (Apr. 6, 2008), pp. 155-164, XP058192632. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022835, 12 pages, Jun. 10,

2020. cited by applicant
International Search Report and Written Opinion for PCT/US2020/022851, 12 pages, Jun. 29, 2020. cited by applicant
Li Bingchao et al: "Elastic-Cache: GPU Cache Architecture for Efficient Fine- and Coarse-Grained Cache-Line Management", 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, May 29, 2017 (May 29, 2017), pp. 82-91, XP033113921. cited by applicant
Goodfellow, et al. "Adaptive Computation and Machine Learning Series", Book, Nov. 18, 2016, pp. 98-165, Chapter 5, The MIT Press, Cambridge, MA. cited by applicant
Ross, et al. "Intel Processor Graphics: Architecture & Programming", Power Point Presentation, Aug. 2015, 78 pages, Intel Corporation, Santa Clara, CA. cited by applicant
Shane Cook, "CUDA Programming", Book, 2013, pp. 37-52, Chapter 3, Elsevier Inc., Amsterdam Netherlands. cited by applicant
Nicholas Wilt, "The CUDA Handbook; A Comprehensive Guide to GPU Programming", Book, Jun. 22, 2013, pp. 41-57, Addison-Wesley Professional, Boston, MA. cited by applicant
Stephen Junkins, "The Compute Architecture of Intel Processor Graphics Gen9", paper, Aug. 14, 2015, 22 pages, Version 1.0, Intel Corporation, Santa Clara, CA. cited by applicant
Non-Final Office Action for U.S. Appl. No. 17/732,308 mailed Jul. 27, 2022, 10 pages. cited by applicant
Office Action for U.S. Appl. No. 18/305,904 Mar. 28, 2025, 25 pages. cited by applicant
Final Office Action in U.S. Appl. No. 18/647,549 mailed Apr. 30, 2025, 7 pages. cited by applicant
Non-Final Office Action issued in U.S. Appl. No. 17/430,611, mailed May 2, 2025, 17 pages. cited by applicant
Office Action for U.S. Appl. No. 18/312,079 mailed Mar. 17, 2025, 15 pages. cited by applicant
Office Action for U.S. Appl. No. 18/490,593, Mar. 24, 2025, 25 pages. cited by applicant
Certificate of Grant for AU Application No. 2020241262, 2 pages, granted on Apr. 24, 2025. cited by applicant
Park, et al., "CNN Deep Learning Acceleration Algorithm for Mobile," Journal of KIIT. vol. 16, No. 10, pp. 1-9, pISSN 1598-8619 (released on Oct. 31, 2019). cited by applicant
Office Action for KR Application No. 10-2021-7025888, Apr. 22, 2025, 8 pages. cited by applicant
Notice of Allowance for CN Application No. 202110214543.3, mailed Feb. 21, 2025, 5 pages. cited by applicant
Notice of Allowance for Korean Application No. 9-5-2025-038947914 mailed Apr. 22, 2025, 4 pages. cited by applicant

---

Primary Examiner: Vo; Tim T

Assistant Examiner: Hasan; Mohammad S

Attorney, Agent or Firm: Jaffery Watson Hamilton & DeSanctis LLP

Background/Summary

CROSS-REFERENCE TO RELATED APPLICATIONS (1) This application is a continuation of U.S. application Ser. No. 17/310,540, filed Aug. 10, 2021, which claims, under 35 U.S.C. § 371, the benefit of and priority to International Application No. PCT/US20/22838, filed Mar. 14, 2020, titled DYNAMIC MEMORY CONFIGURATION, the entire content of which is incorporated herein by reference. International Application No. PCT/US20/22838 is related to and, under 35 U.S.C. 119(e), claims the benefit of and priority to U.S. Provisional Applications 62/819,337, entitled GRAPHICS PROCESSING, by Abhishek Appu, et al., filed Mar. 15, 2019; 62/819,435, entitled GRAPHICS DATA PROCESSING, by Lakshminarayanan Striramassarma, et al., filed Mar. 15, 2019; and 62/819,361, entitled SYSTEMS AND METHODS FOR PARTITIONING CACHE TO REDUCE CACHE ACCESS LATENCY, by Subramaniam Maiyuran, et al., filed Mar. 15, 2019, the contents of all are incorporated herein by reference.

FIELD

(1) This disclosure relates generally to data processing and more particularly to dynamic reconfiguration of memory on a general-purpose graphics processing unit.

BACKGROUND OF THE DISCLOSURE

(2) Current parallel graphics data processing includes systems and methods developed to perform specific operations on graphics data such as, for example, linear interpolation, tessellation, rasterization, texture mapping, depth testing, etc. Traditionally, graphics processors used fixed function computational units to process graphics data; however, more recently, portions of graphics processors have been made programmable, enabling such processors to support a wider variety of operations for processing vertex and fragment data.

(3) To further increase performance, graphics processors typically implement processing techniques such as pipelining that attempt to process, in parallel, as much graphics data as possible throughout the different parts of the graphics pipeline. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. A general overview of software and hardware for SIMT architectures can be found in Shane Cook, CUDA Programming Chapter 3, pages 37-51 (2013).

Description

BRIEF DESCRIPTION OF THE DRAWINGS

(1) The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings in which like references indicate similar elements, and in which:

(2) FIG. **1** is a block diagram illustrating a computer system configured to implement one or more aspects of the embodiments described herein;

(3) FIG. **2**A-**2**D illustrate parallel processor components;

(4) FIG. **3**A-**3**C are block diagrams of graphics multiprocessors and multiprocessor-based GPUs;

(5) FIG. **4**A-**4**F illustrate an exemplary architecture in which a plurality of GPUs is communicatively coupled to a plurality of multi-core processors;

(6) FIG. **5** illustrates a graphics processing pipeline;

(7) FIG. **6** illustrates a machine learning software stack;

(8) FIG. **7** illustrates a general-purpose graphics processing unit;

(9) FIG. **8** illustrates a multi-GPU computing system;

(10) FIG. **9**A-**9**B illustrate layers of exemplary deep neural networks;

(11) FIG. **10** illustrates an exemplary recurrent neural network;

(12) FIG. **11** illustrates training and deployment of a deep neural network;

(13) FIG. **12** is a block diagram illustrating distributed learning;

(14) FIG. **13** illustrates an exemplary inferencing system on a chip (SOC) suitable for performing inferencing using a trained model;

(15) FIG. **14** is a block diagram of a processing system;

(16) FIG. **15**A-**15**C illustrate computing systems and graphics processors;

(17) FIG. **16**A-**16**C illustrate block diagrams of additional graphics processor and compute accelerator architectures;

(18) FIG. **17** is a block diagram of a graphics processing engine of a graphics processor;
(19) FIG. **18**A-**18**B illustrate thread execution logic including an array of processing elements employed in a graphics processor core;
(20) FIG. **19** illustrates an additional execution unit;
(21) FIG. **20** is a block diagram illustrating graphics processor instruction formats;
(22) FIG. **21** is a block diagram of an additional graphics processor architecture;
(23) FIG. **22**A-**22**B illustrate a graphics processor command format and command sequence;
(24) FIG. **23** illustrates exemplary graphics software architecture for a data processing system;
(25) FIG. **24**A is a block diagram illustrating an IP core development system;
(26) FIG. **24**B illustrates a cross-section side view of an integrated circuit package assembly;
(27) FIG. **24**C illustrates a package assembly that includes multiple units of hardware logic chiplets connected to a substrate (e.g., base die);
(28) FIG. **24**D illustrates a package assembly including interchangeable chiplets;
(29) FIG. **25** is a block diagram illustrating an exemplary system on a chip integrated circuit;
(30) FIG. **26**A-**26**B are block diagrams illustrating exemplary graphics processors for use within an SoC;
(31) FIG. **27** illustrates a processing system including a banked cache;
(32) FIG. **28** illustrates a processing system including a cache controller with a dynamic hash unit;
(33) FIG. **29** illustrates a method to enable dynamic reconfiguration of a memory bank hash based on hardware statistics;
(34) FIG. **30** illustrates a heterogenous processing system including a unified memory space;
(35) FIG. **31**A-**31**B illustrates hierarchical page table structures for 4K and 64K pages;
(36) FIG. **32**A-**32**C illustrate a page directory and page tables to enable the mixing of 4K and 64K pages within a hierarchical page table structure, according to an embodiment;
(37) FIG. **33**A-**33**C illustrate methods to enable the mixing of 4K and 64K pages within a hierarchical page table structure;
(38) FIG. **34** illustrates a processing system including a cache having near and far regions;
(39) FIG. **35** illustrates heterogenous processing system having near and far cache regions;
(40) FIG. **36** illustrates a package assembly for a parallel processor including a multi-region L3 cache;
(41) FIG. **37** illustrate a method of managing a cache having multiple cache regions;
(42) FIG. **38** is a block diagram of a data processing system **3800**, according to an embodiment; and
(43) FIG. **39** is a block diagram of a computing device including a graphics processor, according to an embodiment.

DETAILED DESCRIPTION

(44) A graphics processing unit (GPU) is communicatively coupled to host/processor cores to accelerate, for example, graphics operations, machine-learning operations, pattern analysis operations, and/or various general-purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or another interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). Alternatively, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.
(45) Embodiments described herein provide techniques to enable the dynamic reconfiguration of memory on a general-purpose graphics processing unit. One embodiment described herein enables dynamic reconfiguration of cache memory bank assignments based on hardware statistics. One embodiment enables for virtual memory address translation using mixed four kilobyte and 64

kilobyte pages within the same page table hierarchy and under the same page directory. One embodiment provides for a graphics processor and associated heterogenous processing system having near and far regions of the same level of a cache hierarchy.

(46) In the following description, numerous specific details are set forth to provide a more thorough understanding. However, it will be apparent to one of skill in the art that the embodiments described herein may be practiced without one or more of these specific details. In other instances, well-known features have not been described to avoid obscuring the details of the present embodiments.

System Overview

(47) FIG. **1** is a block diagram illustrating a computing system **100** configured to implement one or more aspects of the embodiments described herein. The computing system **100** includes a processing subsystem **101** having one or more processor(s) **102** and a system memory **104** communicating via an interconnection path that may include a memory hub **105**. The memory hub **105** may be a separate component within a chipset component or may be integrated within the one or more processor(s) **102**. The memory hub **105** couples with an I/O subsystem **111** via a communication link **106**. The I/O subsystem **111** includes an I/O hub **107** that can enable the computing system **100** to receive input from one or more input device(s) **108**. Additionally, the I/O hub **107** can enable a display controller, which may be included in the one or more processor(s) **102**, to provide outputs to one or more display device(s) **110**A. In one embodiment the one or more display device(s) **110**A coupled with the I/O hub **107** can include a local, internal, or embedded display device.

(48) The processing subsystem **101**, for example, includes one or more parallel processor(s) **112** coupled to memory hub **105** via a bus or other communication link **113**. The communication link **113** may be one of any number of standards-based communication link technologies or protocols, such as, but not limited to PCI Express, or may be a vendor specific communications interface or communications fabric. The one or more parallel processor(s) **112** may form a computationally focused parallel or vector processing system that can include a large number of processing cores and/or processing clusters, such as a many integrated core (MIC) processor. For example, the one or more parallel processor(s) **112** form a graphics processing subsystem that can output pixels to one of the one or more display device(s) **110**A coupled via the I/O hub **107**. The one or more parallel processor(s) **112** can also include a display controller and display interface (not shown) to enable a direct connection to one or more display device(s) **110**B.

(49) Within the I/O subsystem **111**, a system storage unit **114** can connect to the I/O hub **107** to provide a storage mechanism for the computing system **100**. An I/O switch **116** can be used to provide an interface mechanism to enable connections between the I/O hub **107** and other components, such as a network adapter **118** and/or wireless network adapter **119** that may be integrated into the platform, and various other devices that can be added via one or more add-in device(s) **120**. The add-in device(s) **120** may also include, for example, one or more external graphics processor devices and/or compute accelerators. The network adapter **118** can be an Ethernet adapter or another wired network adapter. The wireless network adapter **119** can include one or more of a Wi-Fi, Bluetooth, near field communication (NFC), or other network device that includes one or more wireless radios.

(50) The computing system **100** can include other components not explicitly shown, including USB or other port connections, optical storage drives, video capture devices, and the like, may also be connected to the I/O hub **107**. Communication paths interconnecting the various components in FIG. **1** may be implemented using any suitable protocols, such as PCI (Peripheral Component Interconnect) based protocols (e.g., PCI-Express), or any other bus or point-to-point communication interfaces and/or protocol(s), such as the NVLink high-speed interconnect, or interconnect protocols known in the art.

(51) The one or more parallel processor(s) **112** may incorporate circuitry optimized for graphics

and video processing, including, for example, video output circuitry, and constitutes a graphics processing unit (GPU). Alternatively or additionally, the one or more parallel processor(s) **112** can incorporate circuitry optimized for general purpose processing, while preserving the underlying computational architecture, described in greater detail herein. Components of the computing system **100** may be integrated with one or more other system elements on a single integrated circuit. For example, the one or more parallel processor(s) **112**, memory hub **105**, processor(s) **102**, and I/O hub **107** can be integrated into a system on chip (SoC) integrated circuit. Alternatively, the components of the computing system **100** can be integrated into a single package to form a system in package (SIP) configuration. In one embodiment at least a portion of the components of the computing system **100** can be integrated into a multi-chip module (MCM), which can be interconnected with other multi-chip modules into a modular computing system.

(52) It will be appreciated that the computing system **100** shown herein is illustrative and that variations and modifications are possible. The connection topology, including the number and arrangement of bridges, the number of processor(s) **102**, and the number of parallel processor(s) **112**, may be modified as desired. For instance, system memory **104** can be connected to the processor(s) **102** directly rather than through a bridge, while other devices communicate with system memory **104** via the memory hub **105** and the processor(s) **102**. In other alternative topologies, the parallel processor(s) **112** are connected to the I/O hub **107** or directly to one of the one or more processor(s) **102**, rather than to the memory hub **105**. In other embodiments, the I/O hub **107** and memory hub **105** may be integrated into a single chip. It is also possible that two or more sets of processor(s) **102** are attached via multiple sockets, which can couple with two or more instances of the parallel processor(s) **112**.

(53) Some of the particular components shown herein are optional and may not be included in all implementations of the computing system **100**. For example, any number of add-in cards or peripherals may be supported, or some components may be eliminated. Furthermore, some architectures may use different terminology for components similar to those illustrated in FIG. **1**. For example, the memory hub **105** may be referred to as a Northbridge in some architectures, while the I/O hub **107** may be referred to as a Southbridge.

(54) FIG. **2**A illustrates a parallel processor **200**. The parallel processor **200** may be a GPU, GPGPU or the like as described herein. The various components of the parallel processor **200** may be implemented using one or more integrated circuit devices, such as programmable processors, application specific integrated circuits (ASICs), or field programmable gate arrays (FPGA). The illustrated parallel processor **200** may be the, or one of the parallel processor(s) **112** shown in FIG. **1**.

(55) The parallel processor **200** includes a parallel processing unit **202**. The parallel processing unit includes an I/O unit **204** that enables communication with other devices, including other instances of the parallel processing unit **202**. The I/O unit **204** may be directly connected to other devices. For instance, the I/O unit **204** connects with other devices via the use of a hub or switch interface, such as memory hub **105**. The connections between the memory hub **105** and the I/O unit **204** form a communication link **113**. Within the parallel processing unit **202**, the I/O unit **204** connects with a host interface **206** and a memory crossbar **216**, where the host interface **206** receives commands directed to performing processing operations and the memory crossbar **216** receives commands directed to performing memory operations.

(56) When the host interface **206** receives a command buffer via the I/O unit **204**, the host interface **206** can direct work operations to perform those commands to a front end **208**. In one embodiment the front end **208** couples with a scheduler **210**, which is configured to distribute commands or other work items to a processing cluster array **212**. The scheduler **210** ensures that the processing cluster array **212** is properly configured and in a valid state before tasks are distributed to the processing clusters of the processing cluster array **212**. The scheduler **210** may be implemented via firmware logic executing on a microcontroller. The microcontroller implemented scheduler **210** is

configurable to perform complex scheduling and work distribution operations at coarse and fine granularity, enabling rapid preemption and context switching of threads executing on the processing array **212**. Preferably, the host software can prove workloads for scheduling on the processing array **212** via one of multiple graphics processing doorbells. The workloads can then be automatically distributed across the processing array **212** by the scheduler **210** logic within the scheduler microcontroller.

(57) The processing cluster array **212** can include up to “N” processing clusters (e.g., cluster **214**A, cluster **214**B, through cluster **214**N). Each cluster **214**A-**214**N of the processing cluster array **212** can execute a large number of concurrent threads. The scheduler **210** can allocate work to the clusters **214**A-**214**N of the processing cluster array **212** using various scheduling and/or work distribution algorithms, which may vary depending on the workload arising for each type of program or computation. The scheduling can be handled dynamically by the scheduler **210**, or can be assisted in part by compiler logic during compilation of program logic configured for execution by the processing cluster array **212**. Optionally, different clusters **214**A-**214**N of the processing cluster array **212** can be allocated for processing different types of programs or for performing different types of computations.

(58) The processing cluster array **212** can be configured to perform various types of parallel processing operations. For example, the cluster array **212** is configured to perform general-purpose parallel compute operations. For example, the processing cluster array **212** can include logic to execute processing tasks including filtering of video and/or audio data, performing modeling operations, including physics operations, and performing data transformations.

(59) The processing cluster array **212** is configured to perform parallel graphics processing operations. In such embodiments in which the parallel processor **200** is configured to perform graphics processing operations, the processing cluster array **212** can include additional logic to support the execution of such graphics processing operations, including, but not limited to texture sampling logic to perform texture operations, as well as tessellation logic and other vertex processing logic. Additionally, the processing cluster array **212** can be configured to execute graphics processing related shader programs such as, but not limited to vertex shaders, tessellation shaders, geometry shaders, and pixel shaders. The parallel processing unit **202** can transfer data from system memory via the I/O unit **204** for processing. During processing the transferred data can be stored to on-chip memory (e.g., parallel processor memory **222**) during processing, then written back to system memory.

(60) In embodiments in which the parallel processing unit **202** is used to perform graphics processing, the scheduler **210** may be configured to divide the processing workload into approximately equal sized tasks, to better enable distribution of the graphics processing operations to multiple clusters **214**A-**214**N of the processing cluster array **212**. In some of these embodiments, portions of the processing cluster array **212** can be configured to perform different types of processing. For example a first portion may be configured to perform vertex shading and topology generation, a second portion may be configured to perform tessellation and geometry shading, and a third portion may be configured to perform pixel shading or other screen space operations, to produce a rendered image for display. Intermediate data produced by one or more of the clusters **214**A-**214**N may be stored in buffers to allow the intermediate data to be transmitted between clusters **214**A-**214**N for further processing.

(61) During operation, the processing cluster array **212** can receive processing tasks to be executed via the scheduler **210**, which receives commands defining processing tasks from front end **208**. For graphics processing operations, processing tasks can include indices of data to be processed, e.g., surface (patch) data, primitive data, vertex data, and/or pixel data, as well as state parameters and commands defining how the data is to be processed (e.g., what program is to be executed). The scheduler **210** may be configured to fetch the indices corresponding to the tasks or may receive the indices from the front end **208**. The front end **208** can be configured to ensure the processing

cluster array **212** is configured to a valid state before the workload specified by incoming command buffers (e.g., batch-buffers, push buffers, etc.) is initiated.

(62) Each of the one or more instances of the parallel processing unit **202** can couple with parallel processor memory **222**. The parallel processor memory **222** can be accessed via the memory crossbar **216**, which can receive memory requests from the processing cluster array **212** as well as the I/O unit **204**. The memory crossbar **216** can access the parallel processor memory **222** via a memory interface **218**. The memory interface **218** can include multiple partition units (e.g., partition unit **220**A, partition unit **220**B, through partition unit **220**N) that can each couple to a portion (e.g., memory unit) of parallel processor memory **222**. The number of partition units **220**A-**220**N may be configured to be equal to the number of memory units, such that a first partition unit **220**A has a corresponding first memory unit **224**A, a second partition unit **220**B has a corresponding memory unit **224**B, and an Nth partition unit **220**N has a corresponding Nth memory unit **224**N. In other embodiments, the number of partition units **220**A-**220**N may not be equal to the number of memory devices.

(63) The memory units **224**A-**224**N can include various types of memory devices, including dynamic random-access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. Optionally, the memory units **224**A-**224**N may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM). Persons skilled in the art will appreciate that the specific implementation of the memory units **224**A-**224**N can vary, and can be selected from one of various conventional designs. Render targets, such as frame buffers or texture maps may be stored across the memory units **224**A-**224**N, allowing partition units **220**A-**220**N to write portions of each render target in parallel to efficiently use the available bandwidth of parallel processor memory **222**. In some embodiments, a local instance of the parallel processor memory **222** may be excluded in favor of a unified memory design that utilizes system memory in conjunction with local cache memory.

(64) Optionally, any one of the clusters **214**A-**214**N of the processing cluster array **212** has the ability to process data that will be written to any of the memory units **224**A-**224**N within parallel processor memory **222**. The memory crossbar **216** can be configured to transfer the output of each cluster **214**A-**214**N to any partition unit **220**A-**220**N or to another cluster **214**A-**214**N, which can perform additional processing operations on the output. Each cluster **214**A-**214**N can communicate with the memory interface **218** through the memory crossbar **216** to read from or write to various external memory devices. In one of the embodiments with the memory crossbar **216** the memory crossbar **216** has a connection to the memory interface **218** to communicate with the I/O unit **204**, as well as a connection to a local instance of the parallel processor memory **222**, enabling the processing units within the different processing clusters **214**A-**214**N to communicate with system memory or other memory that is not local to the parallel processing unit **202**. Generally, the memory crossbar **216** may, for example, by able to use virtual channels to separate traffic streams between the clusters **214**A-**214**N and the partition units **220**A-**220**N.

(65) While a single instance of the parallel processing unit **202** is illustrated within the parallel processor **200**, any number of instances of the parallel processing unit **202** can be included. For example, multiple instances of the parallel processing unit **202** can be provided on a single add-in card, or multiple add-in cards can be interconnected. The different instances of the parallel processing unit **202** can be configured to inter-operate even if the different instances have different numbers of processing cores, different amounts of local parallel processor memory, and/or other configuration differences. Optionally, some instances of the parallel processing unit **202** can include higher precision floating point units relative to other instances. Systems incorporating one or more instances of the parallel processing unit **202** or the parallel processor **200** can be implemented in a variety of configurations and form factors, including but not limited to desktop, laptop, or handheld personal computers, servers, workstations, game consoles, and/or embedded

systems.
(66) FIG. **2**B is a block diagram of a partition unit **220**. The partition unit **220** may be an instance of one of the partition units **220**A-**220**N of FIG. **2**A. As illustrated, the partition unit **220** includes an L2 cache **221**, a frame buffer interface **225**, and a ROP **226** (raster operations unit). The L2 cache **221** is a read/write cache that is configured to perform load and store operations received from the memory crossbar **216** and ROP **226**. Read misses and urgent write-back requests are output by L2 cache **221** to frame buffer interface **225** for processing. Updates can also be sent to the frame buffer via the frame buffer interface **225** for processing. In one embodiment the frame buffer interface **225** interfaces with one of the memory units in parallel processor memory, such as the memory units **224**A-**224**N of FIG. **2**A (e.g., within parallel processor memory **222**). The partition unit **220** may additionally or alternatively also interface with one of the memory units in parallel processor memory via a memory controller (not shown).
(67) In graphics applications, the ROP **226** is a processing unit that performs raster operations such as stencil, z test, blending, and the like. The ROP **226** then outputs processed graphics data that is stored in graphics memory. In some embodiments the ROP **226** includes compression logic to compress depth or color data that is written to memory and decompress depth or color data that is read from memory. The compression logic can be lossless compression logic that makes use of one or more of multiple compression algorithms. The type of compression that is performed by the ROP **226** can vary based on the statistical characteristics of the data to be compressed. For example, in one embodiment, delta color compression is performed on depth and color data on a per-tile basis.
(68) The ROP **226** may be included within each processing cluster (e.g., cluster **214**A-**214**N of FIG. **2**A) instead of within the partition unit **220**. In such embodiment, read and write requests for pixel data are transmitted over the memory crossbar **216** instead of pixel fragment data. The processed graphics data may be displayed on a display device, such as one of the one or more display device(s) **110**A-**110**B of FIG. **1**, routed for further processing by the processor(s) **102**, or routed for further processing by one of the processing entities within the parallel processor **200** of FIG. **2**A.
(69) FIG. **2**C is a block diagram of a processing cluster **214** within a parallel processing unit. For example, the processing cluster is an instance of one of the processing clusters **214**A-**214**N of FIG. **2**A. The processing cluster **214** can be configured to execute many threads in parallel, where the term “thread” refers to an instance of a particular program executing on a particular set of input data. Optionally, single-instruction, multiple-data (SIMD) instruction issue techniques may be used to support parallel execution of a large number of threads without providing multiple independent instruction units. Alternatively, single-instruction, multiple-thread (SIMT) techniques may be used to support parallel execution of a large number of generally synchronized threads, using a common instruction unit configured to issue instructions to a set of processing engines within each one of the processing clusters. Unlike a SIMD execution regime, where all processing engines typically execute identical instructions, SIMT execution allows different threads to more readily follow divergent execution paths through a given thread program. Persons skilled in the art will understand that a SIMD processing regime represents a functional subset of a SIMT processing regime.
(70) Operation of the processing cluster **214** can be controlled via a pipeline manager **232** that distributes processing tasks to SIMT parallel processors. The pipeline manager **232** receives instructions from the scheduler **210** of FIG. **2**A and manages execution of those instructions via a graphics multiprocessor **234** and/or a texture unit **236**. The illustrated graphics multiprocessor **234** is an exemplary instance of a SIMT parallel processor. However, various types of SIMT parallel processors of differing architectures may be included within the processing cluster **214**. One or more instances of the graphics multiprocessor **234** can be included within a processing cluster **214**. The graphics multiprocessor **234** can process data and a data crossbar **240** can be used to distribute

the processed data to one of multiple possible destinations, including other shader units. The pipeline manager **232** can facilitate the distribution of processed data by specifying destinations for processed data to be distributed via the data crossbar **240**.
(71) Each graphics multiprocessor **234** within the processing cluster **214** can include an identical set of functional execution logic (e.g., arithmetic logic units, load-store units, etc.). The functional execution logic can be configured in a pipelined manner in which new instructions can be issued before previous instructions are complete. The functional execution logic supports a variety of operations including integer and floating-point arithmetic, comparison operations, Boolean operations, bit-shifting, and computation of various algebraic functions. The same functional-unit hardware could be leveraged to perform different operations and any combination of functional units may be present.
(72) The instructions transmitted to the processing cluster **214** constitutes a thread. A set of threads executing across the set of parallel processing engines is a thread group. A thread group executes the same program on different input data. Each thread within a thread group can be assigned to a different processing engine within a graphics multiprocessor **234**. A thread group may include fewer threads than the number of processing engines within the graphics multiprocessor **234**. When a thread group includes fewer threads than the number of processing engines, one or more of the processing engines may be idle during cycles in which that thread group is being processed. A thread group may also include more threads than the number of processing engines within the graphics multiprocessor **234**. When the thread group includes more threads than the number of processing engines within the graphics multiprocessor **234**, processing can be performed over consecutive clock cycles. Optionally, multiple thread groups can be executed concurrently on the graphics multiprocessor **234**.
(73) The graphics multiprocessor **234** may include an internal cache memory to perform load and store operations. Optionally, the graphics multiprocessor **234** can forego an internal cache and use a cache memory (e.g., L1 cache **248**) within the processing cluster **214**. Each graphics multiprocessor **234** also has access to L2 caches within the partition units (e.g., partition units **220**A-**220**N of FIG. **2**A) that are shared among all processing clusters **214** and may be used to transfer data between threads. The graphics multiprocessor **234** may also access off-chip global memory, which can include one or more of local parallel processor memory and/or system memory. Any memory external to the parallel processing unit **202** may be used as global memory. Embodiments in which the processing cluster **214** includes multiple instances of the graphics multiprocessor **234** can share common instructions and data, which may be stored in the L1 cache **248**.
(74) Each processing cluster **214** may include an MMU **245** (memory management unit) that is configured to map virtual addresses into physical addresses. In other embodiments, one or more instances of the MMU **245** may reside within the memory interface **218** of FIG. **2**A. The MMU **245** includes a set of page table entries (PTEs) used to map a virtual address to a physical address of a tile and optionally a cache line index. The MMU **245** may include address translation lookaside buffers (TLB) or caches that may reside within the graphics multiprocessor **234** or the L1 cache or processing cluster **214**. The physical address is processed to distribute surface data access locality to allow efficient request interleaving among partition units. The cache line index may be used to determine whether a request for a cache line is a hit or miss.
(75) In graphics and computing applications, a processing cluster **214** may be configured such that each graphics multiprocessor **234** is coupled to a texture unit **236** for performing texture mapping operations, e.g., determining texture sample positions, reading texture data, and filtering the texture data. Texture data is read from an internal texture L1 cache (not shown) or in some embodiments from the L1 cache within graphics multiprocessor **234** and is fetched from an L2 cache, local parallel processor memory, or system memory, as needed. Each graphics multiprocessor **234** outputs processed tasks to the data crossbar **240** to provide the processed task to another processing cluster **214** for further processing or to store the processed task in an L2 cache, local parallel

processor memory, or system memory via the memory crossbar **216**. A preROP **242** (pre-raster operations unit) is configured to receive data from graphics multiprocessor **234**, direct data to ROP units, which may be located with partition units as described herein (e.g., partition units **220**A-**220**N of FIG. **2**A). The preROP **242** unit can perform optimizations for color blending, organize pixel color data, and perform address translations.
(76) It will be appreciated that the core architecture described herein is illustrative and that variations and modifications are possible. Any number of processing units, e.g., graphics multiprocessor **234**, texture units **236**, preROPs **242**, etc., may be included within a processing cluster **214**. Further, while only one processing cluster **214** is shown, a parallel processing unit as described herein may include any number of instances of the processing cluster **214**. Optionally, each processing cluster **214** can be configured to operate independently of other processing clusters **214** using separate and distinct processing units, L1 caches, etc.
(77) FIG. **2**D shows an example of the graphics multiprocessor **234** in which the graphics multiprocessor **234** couples with the pipeline manager **232** of the processing cluster **214**. The graphics multiprocessor **234** has an execution pipeline including but not limited to an instruction cache **252**, an instruction unit **254**, an address mapping unit **256**, a register file **258**, one or more general purpose graphics processing unit (GPGPU) cores **262**, and one or more load/store units **266**. The GPGPU cores **262** and load/store units **266** are coupled with cache memory **272** and shared memory **270** via a memory and cache interconnect **268**. The graphics multiprocessor **234** may additionally include tensor and/or ray-tracing cores **263** that include hardware logic to accelerate matrix and/or ray-tracing operations.
(78) The instruction cache **252** may receive a stream of instructions to execute from the pipeline manager **232**. The instructions are cached in the instruction cache **252** and dispatched for execution by the instruction unit **254**. The instruction unit **254** can dispatch instructions as thread groups (e.g., warps), with each thread of the thread group assigned to a different execution unit within GPGPU core **262**. An instruction can access any of a local, shared, or global address space by specifying an address within a unified address space. The address mapping unit **256** can be used to translate addresses in the unified address space into a distinct memory address that can be accessed by the load/store units **266**.
(79) The register file **258** provides a set of registers for the functional units of the graphics multiprocessor **234**. The register file **258** provides temporary storage for operands connected to the data paths of the functional units (e.g., GPGPU cores **262**, load/store units **266**) of the graphics multiprocessor **234**. The register file **258** may be divided between each of the functional units such that each functional unit is allocated a dedicated portion of the register file **258**. For example, the register file **258** may be divided between the different warps being executed by the graphics multiprocessor **234**.
(80) The GPGPU cores **262** can each include floating point units (FPUs) and/or integer arithmetic logic units (ALUs) that are used to execute instructions of the graphics multiprocessor **234**. In some implementations, the GPGPU cores **262** can include hardware logic that may otherwise reside within the tensor and/or ray-tracing cores **263**. The GPGPU cores **262** can be similar in architecture or can differ in architecture. For example and in one embodiment, a first portion of the GPGPU cores **262** include a single precision FPU and an integer ALU while a second portion of the GPGPU cores include a double precision FPU. Optionally, the FPUs can implement the IEEE 754-2008 standard for floating point arithmetic or enable variable precision floating point arithmetic. The graphics multiprocessor **234** can additionally include one or more fixed function or special function units to perform specific functions such as copy rectangle or pixel blending operations. One or more of the GPGPU cores can also include fixed or special function logic.
(81) The GPGPU cores **262** may include SIMD logic capable of performing a single instruction on multiple sets of data. Optionally, GPGPU cores **262** can physically execute SIMD4, SIMD8, and SIMD16 instructions and logically execute SIMD1, SIMD2, and SIMD32 instructions. The SIMD

instructions for the GPGPU cores can be generated at compile time by a shader compiler or automatically generated when executing programs written and compiled for single program multiple data (SPMD) or SIMT architectures. Multiple threads of a program configured for the SIMT execution model can be executed via a single SIMD instruction. For example and in one embodiment, eight SIMT threads that perform the same or similar operations can be executed in parallel via a single SIMD8 logic unit.
(82) The memory and cache interconnect **268** is an interconnect network that connects each of the functional units of the graphics multiprocessor **234** to the register file **258** and to the shared memory **270**. For example, the memory and cache interconnect **268** is a crossbar interconnect that allows the load/store unit **266** to implement load and store operations between the shared memory **270** and the register file **258**. The register file **258** can operate at the same frequency as the GPGPU cores **262**, thus data transfer between the GPGPU cores **262** and the register file **258** is very low latency. The shared memory **270** can be used to enable communication between threads that execute on the functional units within the graphics multiprocessor **234**. The cache memory **272** can be used as a data cache for example, to cache texture data communicated between the functional units and the texture unit **236**. The shared memory **270** can also be used as a program managed cached. Threads executing on the GPGPU cores **262** can programmatically store data within the shared memory in addition to the automatically cached data that is stored within the cache memory **272**.
(83) FIG. **3**A-**3**C illustrate additional graphics multiprocessors, according to embodiments. FIG. **3**A-**3**B illustrate graphics multiprocessors **325**, **350**, which are related to the graphics multiprocessor **234** of FIG. **2**C and may be used in place of one of those. Therefore, the disclosure of any features in combination with the graphics multiprocessor **234** herein also discloses a corresponding combination with the graphics multiprocessor(s) **325**, **350**, but is not limited to such. FIG. **3**C illustrates a graphics processing unit (GPU) **380** which includes dedicated sets of graphics processing resources arranged into multi-core groups **365**A-**365**N, which correspond to the graphics multiprocessors **325**, **350**. The illustrated graphics multiprocessors **325**, **350** and the multi-core groups **365**A-**365**N can be streaming multiprocessors (SM) capable of simultaneous execution of a large number of execution threads.
(84) The graphics multiprocessor **325** of FIG. **3**A includes multiple additional instances of execution resource units relative to the graphics multiprocessor **234** of FIG. **2**D. For example, the graphics multiprocessor **325** can include multiple instances of the instruction unit **332**A-**332**B, register file **334**A-**334**B, and texture unit(s) **344**A-**344**B. The graphics multiprocessor **325** also includes multiple sets of graphics or compute execution units (e.g., GPGPU core **336**A-**336**B, tensor core **337**A-**337**B, ray-tracing core **338**A-**338**B) and multiple sets of load/store units **340**A-**340**B. The execution resource units have a common instruction cache **330**, texture and/or data cache memory **342**, and shared memory **346**.
(85) The various components can communicate via an interconnect fabric **327**. The interconnect fabric **327** may include one or more crossbar switches to enable communication between the various components of the graphics multiprocessor **325**. The interconnect fabric **327** may be a separate, high-speed network fabric layer upon which each component of the graphics multiprocessor **325** is stacked. The components of the graphics multiprocessor **325** communicate with remote components via the interconnect fabric **327**. For example, the GPGPU cores **336**A-**336**B, **337**A-**337**B, and **3378**A-**338**B can each communicate with shared memory **346** via the interconnect fabric **327**. The interconnect fabric **327** can arbitrate communication within the graphics multiprocessor **325** to ensure a fair bandwidth allocation between components.
(86) The graphics multiprocessor **350** of FIG. **3**B includes multiple sets of execution resources **356**A-**356**D, where each set of execution resource includes multiple instruction units, register files, GPGPU cores, and load store units, as illustrated in FIG. **2**D and FIG. **3**A. The execution resources **356**A-**356**D can work in concert with texture unit(s) **360**A-**360**D for texture operations, while

sharing an instruction cache **354**, and shared memory **353**. For example, the execution resources **356**A-**356**D can share an instruction cache **354** and shared memory **353**, as well as multiple instances of a texture and/or data cache memory **358**A-**358**B. The various components can communicate via an interconnect fabric **352** similar to the interconnect fabric **327** of FIG. **3**A.

(87) Persons skilled in the art will understand that the architecture described in FIGS. **1**, **2**A-**2**D, and **3**A-**3**B are descriptive and not limiting as to the scope of the present embodiments. Thus, the techniques described herein may be implemented on any properly configured processing unit, including, without limitation, one or more mobile application processors, one or more desktop or server central processing units (CPUs) including multi-core CPUs, one or more parallel processing units, such as the parallel processing unit **202** of FIG. **2**A, as well as one or more graphics processors or special purpose processing units, without departure from the scope of the embodiments described herein.

(88) The parallel processor or GPGPU as described herein may be communicatively coupled to host/processor cores to accelerate graphics operations, machine-learning operations, pattern analysis operations, and various general-purpose GPU (GPGPU) functions. The GPU may be communicatively coupled to the host processor/cores over a bus or other interconnect (e.g., a high-speed interconnect such as PCIe or NVLink). In other embodiments, the GPU may be integrated on the same package or chip as the cores and communicatively coupled to the cores over an internal processor bus/interconnect (i.e., internal to the package or chip). Regardless of the manner in which the GPU is connected, the processor cores may allocate work to the GPU in the form of sequences of commands/instructions contained in a work descriptor. The GPU then uses dedicated circuitry/logic for efficiently processing these commands/instructions.

(89) FIG. **3**C illustrates a graphics processing unit (GPU) **380** which includes dedicated sets of graphics processing resources arranged into multi-core groups **365**A-**365**N. While the details of only a single multi-core group **365**A are provided, it will be appreciated that the other multi-core groups **365**B-**365**N may be equipped with the same or similar sets of graphics processing resources. Details described with respect to the multi-core groups **365**A-**365**N may also apply to any graphics multiprocessor **234**, **325**, **350** described herein.

(90) As illustrated, a multi-core group **365**A may include a set of graphics cores **370**, a set of tensor cores **371**, and a set of ray tracing cores **372**. A scheduler/dispatcher **368** schedules and dispatches the graphics threads for execution on the various cores **370**, **371**, **372**. A set of register files **369** store operand values used by the cores **370**, **371**, **372** when executing the graphics threads. These may include, for example, integer registers for storing integer values, floating point registers for storing floating point values, vector registers for storing packed data elements (integer and/or floating-point data elements) and tile registers for storing tensor/matrix values. The tile registers may be implemented as combined sets of vector registers.

(91) One or more combined level 1 (L1) caches and shared memory **373** store graphics data such as texture data, vertex data, pixel data, ray data, bounding volume data, etc., locally within each multi-core group **365**A. One or more texture units **374** can also be used to perform texturing operations, such as texture mapping and sampling. A Level 2 (L2) cache **375** shared by all or a subset of the multi-core groups **365**A-**365**N stores graphics data and/or instructions for multiple concurrent graphics threads. As illustrated, the L2 cache **375** may be shared across a plurality of multi-core groups **365**A-**365**N. One or more memory controllers **367** couple the GPU **380** to a memory **366** which may be a system memory (e.g., DRAM) and/or a dedicated graphics memory (e.g., GDDR6 memory).

(92) Input/output (I/O) circuitry **363** couples the GPU **380** to one or more I/O devices **362** such as digital signal processors (DSPs), network controllers, or user input devices. An on-chip interconnect may be used to couple the I/O devices **362** to the GPU **380** and memory **366**. One or more I/O memory management units (IOMMUs) **364** of the I/O circuitry **363** couple the I/O devices **362** directly to the system memory **366**. Optionally, the IOMMU **364** manages multiple

sets of page tables to map virtual addresses to physical addresses in system memory **366**. The I/O devices **362**, CPU(s) **361**, and GPU(s) **380** may then share the same virtual address space.
(93) In one implementation of the IOMMU **364**, the IOMMU **364** supports virtualization. In this case, it may manage a first set of page tables to map guest/graphics virtual addresses to guest/graphics physical addresses and a second set of page tables to map the guest/graphics physical addresses to system/host physical addresses (e.g., within system memory **366**). The base addresses of each of the first and second sets of page tables may be stored in control registers and swapped out on a context switch (e.g., so that the new context is provided with access to the relevant set of page tables). While not illustrated in FIG. 3C, each of the cores **370**, **371**, **372** and/or multi-core groups **365**A-**365**N may include translation lookaside buffers (TLBs) to cache guest virtual to guest physical translations, guest physical to host physical translations, and guest virtual to host physical translations.
(94) The CPUs **361**, GPUs **380**, and I/O devices **362** may be integrated on a single semiconductor chip and/or chip package. The illustrated memory **366** may be integrated on the same chip or may be coupled to the memory controllers **367** via an off-chip interface. In one implementation, the memory **366** comprises GDDR6 memory which shares the same virtual address space as other physical system-level memories, although the underlying principles described herein are not limited to this specific implementation.
(95) The tensor cores **371** may include a plurality of execution units specifically designed to perform matrix operations, which are the fundamental compute operation used to perform deep learning operations. For example, simultaneous matrix multiplication operations may be used for neural network training and inferencing. The tensor cores **371** may perform matrix processing using a variety of operand precisions including single precision floating-point (e.g., 32 bits), half-precision floating point (e.g., 16 bits), integer words (16 bits), bytes (8 bits), and half-bytes (4 bits). For example, a neural network implementation extracts features of each rendered scene, potentially combining details from multiple frames, to construct a high-quality final image.
(96) In deep learning implementations, parallel matrix multiplication work may be scheduled for execution on the tensor cores **371**. The training of neural networks, in particular, requires a significant number of matrix dot product operations. In order to process an inner-product formulation of an N×N×N matrix multiply, the tensor cores **371** may include at least N dot-product processing elements. Before the matrix multiply begins, one entire matrix is loaded into tile registers and at least one column of a second matrix is loaded each cycle for N cycles. Each cycle, there are N dot products that are processed.
(97) Matrix elements may be stored at different precisions depending on the particular implementation, including 16-bit words, 8-bit bytes (e.g., INT8) and 4-bit half-bytes (e.g., INT4). Different precision modes may be specified for the tensor cores **371** to ensure that the most efficient precision is used for different workloads (e.g., such as inferencing workloads which can tolerate quantization to bytes and half-bytes).
(98) The ray tracing cores **372** may accelerate ray tracing operations for both real-time ray tracing and non-real-time ray tracing implementations. In particular, the ray tracing cores **372** may include ray traversal/intersection circuitry for performing ray traversal using bounding volume hierarchies (BVHs) and identifying intersections between rays and primitives enclosed within the BVH volumes. The ray tracing cores **372** may also include circuitry for performing depth testing and culling (e.g., using a Z buffer or similar arrangement). In one implementation, the ray tracing cores **372** perform traversal and intersection operations in concert with the image denoising techniques described herein, at least a portion of which may be executed on the tensor cores **371**. For example, the tensor cores **371** may implement a deep learning neural network to perform denoising of frames generated by the ray tracing cores **372**. However, the CPU(s) **361**, graphics cores **370**, and/or ray tracing cores **372** may also implement all or a portion of the denoising and/or deep learning algorithms.

(99) In addition, as described above, a distributed approach to denoising may be employed in which the GPU **380** is in a computing device coupled to other computing devices over a network or high-speed interconnect. In this distributed approach, the interconnected computing devices may share neural network learning/training data to improve the speed with which the overall system learns to perform denoising for different types of image frames and/or different graphics applications.
(100) The ray tracing cores **372** may process all BVH traversal and/or ray-primitive intersections, saving the graphics cores **370** from being overloaded with thousands of instructions per ray. For example, each ray tracing core **372** includes a first set of specialized circuitry for performing bounding box tests (e.g., for traversal operations) and/or a second set of specialized circuitry for performing the ray-triangle intersection tests (e.g., intersecting rays which have been traversed). Thus, for example, the multi-core group **365**A can simply launch a ray probe, and the ray tracing cores **372** independently perform ray traversal and intersection and return hit data (e.g., a hit, no hit, multiple hits, etc.) to the thread context. The other cores **370**, **371** are freed to perform other graphics or compute work while the ray tracing cores **372** perform the traversal and intersection operations.
(101) Optionally, each ray tracing core **372** may include a traversal unit to perform BVH testing operations and/or an intersection unit which performs ray-primitive intersection tests. The intersection unit generates a "hit", "no hit", or "multiple hit" response, which it provides to the appropriate thread. During the traversal and intersection operations, the execution resources of the other cores (e.g., graphics cores **370** and tensor cores **371**) are freed to perform other forms of graphics work.
(102) In one optional embodiment described below, a hybrid rasterization/ray tracing approach is used in which work is distributed between the graphics cores **370** and ray tracing cores **372**.
(103) The ray tracing cores **372** (and/or other cores **370**, **371**) may include hardware support for a ray tracing instruction set such as Microsoft's DirectX Ray Tracing (DXR) which includes a DispatchRays command, as well as ray-generation, closest-hit, any-hit, and miss shaders, which enable the assignment of unique sets of shaders and textures for each object. Another ray tracing platform which may be supported by the ray tracing cores **372**, graphics cores **370** and tensor cores **371** is Vulkan 1.1.85. Note, however, that the underlying principles described herein are not limited to any particular ray tracing ISA.
(104) In general, the various cores **372**, **371**, **370** may support a ray tracing instruction set that includes instructions/functions for one or more of ray generation, closest hit, any hit, ray-primitive intersection, per-primitive and hierarchical bounding box construction, miss, visit, and exceptions. More specifically, a preferred embodiment includes ray tracing instructions to perform one or more of the following functions:
(105) Ray Generation—Ray generation instructions may be executed for each pixel, sample, or other user-defined work assignment.
(106) Closest Hit—A closest hit instruction may be executed to locate the closest intersection point of a ray with primitives within a scene.
(107) Any Hit—An any hit instruction identifies multiple intersections between a ray and primitives within a scene, potentially to identify a new closest intersection point.
(108) Intersection—An intersection instruction performs a ray-primitive intersection test and outputs a result.
(109) Per-primitive Bounding box Construction—This instruction builds a bounding box around a given primitive or group of primitives (e.g., when building a new BVH or other acceleration data structure).
(110) Miss—Indicates that a ray misses all geometry within a scene, or specified region of a scene.
(111) Visit—Indicates the children volumes a ray will traverse.
(112) Exceptions—Includes various types of exception handlers (e.g., invoked for various error

conditions).
(113) Techniques for GPU to Host Processor Interconnection
(114) FIG. **4**A illustrates an exemplary architecture in which a plurality of GPUs **410-413**, e.g. such as the parallel processors **200** shown in FIG. **2**A, are communicatively coupled to a plurality of multi-core processors **405-406** over high-speed links **440**A-**440**D (e.g., buses, point-to-point interconnects, etc.). The high-speed links **440**A-**440**D may support a communication throughput of 4 GB/s, 30 GB/s, 80 GB/s or higher, depending on the implementation. Various interconnect protocols may be used including, but not limited to, PCIe 4.0 or 5.0 and NVLink 2.0. However, the underlying principles described herein are not limited to any particular communication protocol or throughput.
(115) Two or more of the GPUs **410-413** may be interconnected over high-speed links **442**A-**442**B, which may be implemented using the same or different protocols/links than those used for high-speed links **440**A-**440**D. Similarly, two or more of the multi-core processors **405-406** may be connected over high speed link **443** which may be symmetric multi-processor (SMP) buses operating at 20 GB/s, 30 GB/s, 120 GB/s or higher. Alternatively, all communication between the various system components shown in FIG. **4**A may be accomplished using the same protocols/links (e.g., over a common interconnection fabric). As mentioned, however, the underlying principles described herein are not limited to any particular type of interconnect technology.
(116) Multi-core processor **405** and multi-core processor **406** may be communicatively coupled to a processor memory **401-402**, via memory interconnects **430**A-**430**B, respectively, and each GPU **410-413** is communicatively coupled to GPU memory **420-423** over GPU memory interconnects **450**A-**450**D, respectively. The memory interconnects **430**A-**430**B and **450**A-**450**D may utilize the same or different memory access technologies. By way of example, and not limitation, the processor memories **401-402** and GPU memories **420-423** may be volatile memories such as dynamic random-access memories (DRAMs) (including stacked DRAMs), Graphics DDR SDRAM (GDDR) (e.g., GDDR5, GDDR6), or High Bandwidth Memory (HBM) and/or may be non-volatile memories such as 3D XPoint/Optane or Nano-Ram. For example, some portion of the memories may be volatile memory and another portion may be non-volatile memory (e.g., using a two-level memory (2LM) hierarchy).
(117) As described below, although the various processors **405-406** and GPUs **410-413** may be physically coupled to a particular memory **401-402**, **420-423**, respectively, a unified memory architecture may be implemented in which the same virtual system address space (also referred to as the “effective address” space) is distributed among all of the various physical memories. For example, processor memories **401-402** may each comprise 64 GB of the system memory address space and GPU memories **420-423** may each comprise 32 GB of the system memory address space (resulting in a total of 256 GB addressable memory in this example).
(118) FIG. **4**B illustrates additional optional details for an interconnection between a multi-core processor **407** and a graphics acceleration module **446**. The graphics acceleration module **446** may include one or more GPU chips integrated on a line card which is coupled to the processor **407** via the high-speed link **440**. Alternatively, the graphics acceleration module **446** may be integrated on the same package or chip as the processor **407**.
(119) The illustrated processor **407** includes a plurality of cores **460**A-**460**D, each with a translation lookaside buffer **461**A-**461**D and one or more caches **462**A-**462**D. The cores may include various other components for executing instructions and processing data which are not illustrated to avoid obscuring the underlying principles of the components described herein (e.g., instruction fetch units, branch prediction units, decoders, execution units, reorder buffers, etc.). The caches **462**A-**462**D may comprise level 1 (L1) and level 2 (L2) caches. In addition, one or more shared caches **456** may be included in the caching hierarchy and shared by sets of the cores **460**A-**460**D. For example, one embodiment of the processor **407** includes 24 cores, each with its own L1 cache, twelve shared L2 caches, and twelve shared L3 caches. In this embodiment, one of the L2 and L3

caches are shared by two adjacent cores. The processor **407** and the graphics accelerator integration module **446** connect with system memory **441**, which may include processor memories **401-402**.
(120) Coherency is maintained for data and instructions stored in the various caches **462**A-**462**D, **456** and system memory **441** via inter-core communication over a coherence bus **464**. For example, each cache may have cache coherency logic/circuitry associated therewith to communicate to over the coherence bus **464** in response to detected reads or writes to particular cache lines. In one implementation, a cache snooping protocol is implemented over the coherence bus **464** to snoop cache accesses. Cache snooping/coherency techniques are well understood by those of skill in the art and will not be described in detail here to avoid obscuring the underlying principles described herein.
(121) A proxy circuit **425** may be provided that communicatively couples the graphics acceleration module **446** to the coherence bus **464**, allowing the graphics acceleration module **446** to participate in the cache coherence protocol as a peer of the cores. In particular, an interface **435** provides connectivity to the proxy circuit **425** over high-speed link **440** (e.g., a PCIe bus, NVLink, etc.) and an interface **437** connects the graphics acceleration module **446** to the high-speed link **440**.
(122) In one implementation, an accelerator integration circuit **436** provides cache management, memory access, context management, and interrupt management services on behalf of a plurality of graphics processing engines **431**, **432**, N of the graphics acceleration module **446**. The graphics processing engines **431**, **432**, N may each comprise a separate graphics processing unit (GPU). Alternatively, the graphics processing engines **431**, **432**, N may comprise different types of graphics processing engines within a GPU such as graphics execution units, media processing engines (e.g., video encoders/decoders), samplers, and blit engines. In other words, the graphics acceleration module may be a GPU with a plurality of graphics processing engines **431-432**, N or the graphics processing engines **431-432**, N may be individual GPUs integrated on a common package, line card, or chip.
(123) The accelerator integration circuit **436** may include a memory management unit (MMU) **439** for performing various memory management functions such as virtual-to-physical memory translations (also referred to as effective-to-real memory translations) and memory access protocols for accessing system memory **441**. The MMU **439** may also include a translation lookaside buffer (TLB) (not shown) for caching the virtual/effective to physical/real address translations. In one implementation, a cache **438** stores commands and data for efficient access by the graphics processing engines **431-432**, N. The data stored in cache **438** and graphics memories **433-434**, M may be kept coherent with the core caches **462**A-**462**D, **456** and system memory **411**. As mentioned, this may be accomplished via proxy circuit **425** which takes part in the cache coherency mechanism on behalf of cache **438** and memories **433-434**, M (e.g., sending updates to the cache **438** related to modifications/accesses of cache lines on processor caches **462**A-**462**D, **456** and receiving updates from the cache **438**).
(124) A set of registers **445** store context data for threads executed by the graphics processing engines **431-432**, N and a context management circuit **448** manages the thread contexts. For example, the context management circuit **448** may perform save and restore operations to save and restore contexts of the various threads during contexts switches (e.g., where a first thread is saved and a second thread is stored so that the second thread can be execute by a graphics processing engine). For example, on a context switch, the context management circuit **448** may store current register values to a designated region in memory (e.g., identified by a context pointer). It may then restore the register values when returning to the context. An interrupt management circuit **447**, for example, may receive and processes interrupts received from system devices.
(125) In one implementation, virtual/effective addresses from a graphics processing engine **431** are translated to real/physical addresses in system memory **411** by the MMU **439**. Optionally, the accelerator integration circuit **436** supports multiple (e.g., 4, 8, 16) graphics accelerator modules **446** and/or other accelerator devices. The graphics accelerator module **446** may be dedicated to a

single application executed on the processor **407** or may be shared between multiple applications. Optionally, a virtualized graphics execution environment is provided in which the resources of the graphics processing engines **431-432**, N are shared with multiple applications or virtual machines (VMs). The resources may be subdivided into "slices" which are allocated to different VMs and/or applications based on the processing requirements and priorities associated with the VMs and/or applications.

(126) Thus, the accelerator integration circuit **436** acts as a bridge to the system for the graphics acceleration module **446** and provides address translation and system memory cache services. In one embodiment, to facilitate the bridging functionality, the accelerator integration circuit **436** may also include shared I/O **497** (e.g., PCIe, USB) and hardware to enable system control of voltage, clocking, performance, thermals, and security. The shared I/O **497** may utilize separate physical connections or may traverse the high-speed link **440**. In addition, the accelerator integration circuit **436** may provide virtualization facilities for the host processor to manage virtualization of the graphics processing engines, interrupts, and memory management.

(127) Because hardware resources of the graphics processing engines **431-432**, N are mapped explicitly to the real address space seen by the host processor **407**, any host processor can address these resources directly using an effective address value. One optional function of the accelerator integration circuit **436** is the physical separation of the graphics processing engines **431-432**, N so that they appear to the system as independent units.

(128) One or more graphics memories **433-434**, M may be coupled to each of the graphics processing engines **431-432**, N, respectively. The graphics memories **433-434**, M store instructions and data being processed by each of the graphics processing engines **431-432**, N. The graphics memories **433-434**, M may be volatile memories such as DRAMs (including stacked DRAMs), GDDR memory (e.g., GDDR5, GDDR6), or HBM, and/or may be non-volatile memories such as 3D XPoint/Optane or Nano-Ram.

(129) To reduce data traffic over the high-speed link **440**, biasing techniques may be used to ensure that the data stored in graphics memories **433-434**, M is data which will be used most frequently by the graphics processing engines **431-432**, N and preferably not used by the cores **460**A-**460**D (at least not frequently). Similarly, the biasing mechanism attempts to keep data needed by the cores (and preferably not the graphics processing engines **431-432**, N) within the caches **462**A-**462**D, **456** of the cores and system memory **411**.

(130) According to a variant shown in FIG. **4**C the accelerator integration circuit **436** is integrated within the processor **407**. The graphics processing engines **431-432**, N communicate directly over the high-speed link **440** to the accelerator integration circuit **436** via interface **437** and interface **435** (which, again, may be utilize any form of bus or interface protocol). The accelerator integration circuit **436** may perform the same operations as those described with respect to FIG. **4**B, but potentially at a higher throughput given its close proximity to the coherence bus **464** and caches **462**A-**462**D, **456**.

(131) The embodiments described may support different programming models including a dedicated-process programming model (no graphics acceleration module virtualization) and shared programming models (with virtualization). The latter may include programming models which are controlled by the accelerator integration circuit **436** and programming models which are controlled by the graphics acceleration module **446**.

(132) In the embodiments of the dedicated process model, graphics processing engines **431-432**, N may be dedicated to a single application or process under a single operating system. The single application can funnel other application requests to the graphics engines **431-432**, N, providing virtualization within a VM/partition.

(133) In the dedicated-process programming models, the graphics processing engines **431-432**, N, may be shared by multiple VM/application partitions. The shared models require a system hypervisor to virtualize the graphics processing engines **431-432**, N to allow access by each

operating system. For single-partition systems without a hypervisor, the graphics processing engines **431-432**, N are owned by the operating system. In both cases, the operating system can virtualize the graphics processing engines **431-432**, N to provide access to each process or application.

(134) For the shared programming model, the graphics acceleration module **446** or an individual graphics processing engine **431-432**, N selects a process element using a process handle. The process elements may be stored in system memory **411** and be addressable using the effective address to real address translation techniques described herein. The process handle may be an implementation-specific value provided to the host process when registering its context with the graphics processing engine **431-432**, N (that is, calling system software to add the process element to the process element linked list). The lower 16-bits of the process handle may be the offset of the process element within the process element linked list.

(135) FIG. **4**D illustrates an exemplary accelerator integration slice **490**. As used herein, a “slice” comprises a specified portion of the processing resources of the accelerator integration circuit **436**. Application effective address space **482** within system memory **411** stores process elements **483**. The process elements **483** may be stored in response to GPU invocations **481** from applications **480** executed on the processor **407**. A process element **483** contains the process state for the corresponding application **480**. A work descriptor (WD) **484** contained in the process element **483** can be a single job requested by an application or may contain a pointer to a queue of jobs. In the latter case, the WD **484** is a pointer to the job request queue in the application's address space **482**.

(136) The graphics acceleration module **446** and/or the individual graphics processing engines **431-432**, N can be shared by all or a subset of the processes in the system. For example, the technologies described herein may include an infrastructure for setting up the process state and sending a WD **484** to a graphics acceleration module **446** to start a job in a virtualized environment.

(137) In one implementation, the dedicated-process programming model is implementation-specific. In this model, a single process owns the graphics acceleration module **446** or an individual graphics processing engine **431**. Because the graphics acceleration module **446** is owned by a single process, the hypervisor initializes the accelerator integration circuit **436** for the owning partition and the operating system initializes the accelerator integration circuit **436** for the owning process at the time when the graphics acceleration module **446** is assigned.

(138) In operation, a WD fetch unit **491** in the accelerator integration slice **490** fetches the next WD **484** which includes an indication of the work to be done by one of the graphics processing engines of the graphics acceleration module **446**. Data from the WD **484** may be stored in registers **445** and used by the MMU **439**, interrupt management circuit **447** and/or context management circuit **448** as illustrated. For example, the MMU **439** may include segment/page walk circuitry for accessing segment/page tables **486** within the OS virtual address space **485**. The interrupt management circuit **447** may process interrupt events **492** received from the graphics acceleration module **446**. When performing graphics operations, an effective address **493** generated by a graphics processing engine **431-432**, N is translated to a real address by the MMU **439**.

(139) The same set of registers **445** may be duplicated for each graphics processing engine **431-432**, N and/or graphics acceleration module **446** and may be initialized by the hypervisor or operating system. Each of these duplicated registers may be included in an accelerator integration slice **490**. Exemplary registers that may be initialized by the hypervisor are shown in Table 1.

(140) TABLE-US-00001 TABLE 1 Hypervisor Initialized Registers 1 Slice Control Register 2 Real Address (RA) Scheduled Processes Area Pointer 3 Authority Mask Override Register 4 Interrupt Vector Table Entry Offset 5 Interrupt Vector Table Entry Limit 6 State Register 7 Logical Partition ID 8 Real address (RA) Hypervisor Accelerator Utilization Record Pointer 9 Storage Description Register

(141) Exemplary registers that may be initialized by the operating system are shown in Table 2.

(142) TABLE-US-00002 TABLE 2 Operating System Initialized Registers 1 Process and Thread

Identification 2 Effective Address (EA) Context Save/Restore Pointer 3 Virtual Address (VA) Accelerator Utilization Record Pointer 4 Virtual Address (VA) Storage Segment Table Pointer 5 Authority Mask 6 Work descriptor

(143) Each WD **484** may be specific to a particular graphics acceleration module **446** and/or graphics processing engine **431-432**, N. It contains all the information a graphics processing engine **431-432**, N requires to do its work or it can be a pointer to a memory location where the application has set up a command queue of work to be completed.

(144) FIG. **4**E illustrates additional optional details of a shared model. It includes a hypervisor real address space **498** in which a process element list **499** is stored. The hypervisor real address space **498** is accessible via a hypervisor **496** which virtualizes the graphics acceleration module engines for the operating system **495**.

(145) The shared programming models allow for all or a subset of processes from all or a subset of partitions in the system to use a graphics acceleration module **446**. There are two programming models where the graphics acceleration module **446** is shared by multiple processes and partitions: time-sliced shared and graphics directed shared.

(146) In this model, the system hypervisor **496** owns the graphics acceleration module **446** and makes its function available to all operating systems **495**. For a graphics acceleration module **446** to support virtualization by the system hypervisor **496**, the graphics acceleration module **446** may adhere to the following requirements: 1) An application's job request must be autonomous (that is, the state does not need to be maintained between jobs), or the graphics acceleration module **446** must provide a context save and restore mechanism. 2) An application's job request is guaranteed by the graphics acceleration module **446** to complete in a specified amount of time, including any translation faults, or the graphics acceleration module **446** provides the ability to preempt the processing of the job. 3) The graphics acceleration module **446** must be guaranteed fairness between processes when operating in the directed shared programming model.

(147) For the shared model, the application **480** may be required to make an operating system **495** system call with a graphics acceleration module **446** type, a work descriptor (WD), an authority mask register (AMR) value, and a context save/restore area pointer (CSRP). The graphics acceleration module **446** type describes the targeted acceleration function for the system call. The graphics acceleration module **446** type may be a system-specific value. The WD is formatted specifically for the graphics acceleration module **446** and can be in the form of a graphics acceleration module **446** command, an effective address pointer to a user-defined structure, an effective address pointer to a queue of commands, or any other data structure to describe the work to be done by the graphics acceleration module **446**. In one embodiment, the AMR value is the AMR state to use for the current process. The value passed to the operating system is similar to an application setting the AMR. If the accelerator integration circuit **436** and graphics acceleration module **446** implementations do not support a User Authority Mask Override Register (UAMOR), the operating system may apply the current UAMOR value to the AMR value before passing the AMR in the hypervisor call. The hypervisor **496** may optionally apply the current Authority Mask Override Register (AMOR) value before placing the AMR into the process element **483**. The CSRP may be one of the registers **445** containing the effective address of an area in the application's address space **482** for the graphics acceleration module **446** to save and restore the context state. This pointer is optional if no state is required to be saved between jobs or when a job is preempted. The context save/restore area may be pinned system memory.

(148) Upon receiving the system call, the operating system **495** may verify that the application **480** has registered and been given the authority to use the graphics acceleration module **446**. The operating system **495** then calls the hypervisor **496** with the information shown in Table 3.

(149) TABLE-US-00003 TABLE 3 OS to Hypervisor Call Parameters 1 A work descriptor (WD) 2 An Authority Mask Register (AMR) value (potentially masked). 3 An effective address (EA) Context Save/ Restore Area Pointer (CSRP) 4 A process ID (PID) and optional thread ID (TID) 5 A

virtual address (VA) accelerator utilization record pointer (AURP) 6 The virtual address of the storage segment table pointer (SSTP) 7 A logical interrupt service number (LISN)

(150) Upon receiving the hypervisor call, the hypervisor **496** verifies that the operating system **495** has registered and been given the authority to use the graphics acceleration module **446**. The hypervisor **496** then puts the process element **483** into the process element linked list for the corresponding graphics acceleration module **446** type. The process element may include the information shown in Table 4.

(151) TABLE-US-00004 TABLE 4 Process Element Information 1 A work descriptor (WD) 2 An Authority Mask Register (AMR) value (potentially masked). 3 An effective address (EA) Context Save/ Restore Area Pointer (CSRP) 4 A process ID (PID) and optional thread ID (TID) 5 A virtual address (VA) accelerator utilization record pointer (AURP) 6 The virtual address of the storage segment table pointer (SSTP) 7 A logical interrupt service number (LISN) 8 Interrupt vector table, derived from the hypervisor call parameters. 9 A state register (SR) value 10 A logical partition ID (LPID) 11 A real address (RA) hypervisor accelerator utilization record pointer 12 The Storage Descriptor Register (SDR)

(152) The hypervisor may initialize a plurality of accelerator integration slice **490** registers **445**.

(153) As illustrated in FIG. **4**F, in one optional implementation a unified memory addressable via a common virtual memory address space used to access the physical processor memories **401-402** and GPU memories **420-423** is employed. In this implementation, operations executed on the GPUs **410-413** utilize the same virtual/effective memory address space to access the processors memories **401-402** and vice versa, thereby simplifying programmability. A first portion of the virtual/effective address space may be allocated to the processor memory **401**, a second portion to the second processor memory **402**, a third portion to the GPU memory **420**, and so on. The entire virtual/effective memory space (sometimes referred to as the effective address space) may thereby be distributed across each of the processor memories **401-402** and GPU memories **420-423**, allowing any processor or GPU to access any physical memory with a virtual address mapped to that memory.

(154) Bias/coherence management circuitry **494**A-**494**E within one or more of the MMUs **439**A-**439**E may be provided that ensures cache coherence between the caches of the host processors (e.g., **405**) and the GPUs **410-413** and implements biasing techniques indicating the physical memories in which certain types of data should be stored. While multiple instances of bias/coherence management circuitry **494**A-**494**E are illustrated in FIG. **4**F, the bias/coherence circuitry may be implemented within the MMU of one or more host processors **405** and/or within the accelerator integration circuit **436**.

(155) The GPU-attached memory **420-423** may be mapped as part of system memory, and accessed using shared virtual memory (SVM) technology, but without suffering the typical performance drawbacks associated with full system cache coherence. The ability to GPU-attached memory **420-423** to be accessed as system memory without onerous cache coherence overhead provides a beneficial operating environment for GPU offload. This arrangement allows the host processor **405** software to setup operands and access computation results, without the overhead of tradition I/O DMA data copies. Such traditional copies involve driver calls, interrupts and memory mapped I/O (MMIO) accesses that are all inefficient relative to simple memory accesses. At the same time, the ability to access GPU attached memory **420-423** without cache coherence overheads can be critical to the execution time of an offloaded computation. In cases with substantial streaming write memory traffic, for example, cache coherence overhead can significantly reduce the effective write bandwidth seen by a GPU **410-413**. The efficiency of operand setup, the efficiency of results access, and the efficiency of GPU computation all play a role in determining the effectiveness of GPU offload.

(156) A selection of between GPU bias and host processor bias may be driven by a bias tracker data structure. A bias table may be used, for example, which may be a page-granular structure (i.e.,

controlled at the granularity of a memory page) that includes 1 or 2 bits per GPU-attached memory page. The bias table may be implemented in a stolen memory range of one or more GPU-attached memories **420-423**, with or without a bias cache in the GPU **410-413** (e.g., to cache frequently/recently used entries of the bias table). Alternatively, the entire bias table may be maintained within the GPU.

(157) In one implementation, the bias table entry associated with each access to the GPU-attached memory **420-423** is accessed prior the actual access to the GPU memory, causing the following operations. First, local requests from the GPU **410-413** that find their page in GPU bias are forwarded directly to a corresponding GPU memory **420-423**. Local requests from the GPU that find their page in host bias are forwarded to the processor **405** (e.g., over a high-speed link as discussed above). Optionally, requests from the processor **405** that find the requested page in host processor bias complete the request like a normal memory read. Alternatively, requests directed to a GPU-biased page may be forwarded to the GPU **410-413**. The GPU may then transition the page to a host processor bias if it is not currently using the page.

(158) The bias state of a page can be changed either by a software-based mechanism, a hardware-assisted software-based mechanism, or, for a limited set of cases, a purely hardware-based mechanism.

(159) One mechanism for changing the bias state employs an API call (e.g. OpenCL), which, in turn, calls the GPU's device driver which, in turn, sends a message (or enqueues a command descriptor) to the GPU directing it to change the bias state and, for some transitions, perform a cache flushing operation in the host. The cache flushing operation is required for a transition from host processor **405** bias to GPU bias, but is not required for the opposite transition.

(160) Cache coherency may be maintained by temporarily rendering GPU-biased pages uncacheable by the host processor **405**. To access these pages, the processor **405** may request access from the GPU **410** which may or may not grant access right away, depending on the implementation. Thus, to reduce communication between the host processor **405** and GPU **410** it is beneficial to ensure that GPU-biased pages are those which are required by the GPU but not the host processor **405** and vice versa.

(161) Graphics Processing Pipeline

(162) FIG. **5** illustrates a graphics processing pipeline **500**. A graphics multiprocessor, such as graphics multiprocessor **234** as in FIG. **2**D, graphics multiprocessor **325** of FIG. **3**A, graphics multiprocessor **350** of FIG. **3**B can implement the illustrated graphics processing pipeline **500**. The graphics multiprocessor can be included within the parallel processing subsystems as described herein, such as the parallel processor **200** of FIG. **2**A, which may be related to the parallel processor(s) **112** of FIG. **1** and may be used in place of one of those. The various parallel processing systems can implement the graphics processing pipeline **500** via one or more instances of the parallel processing unit (e.g., parallel processing unit **202** of FIG. **2**A) as described herein. For example, a shader unit (e.g., graphics multiprocessor **234** of FIG. **2**C) may be configured to perform the functions of one or more of a vertex processing unit **504**, a tessellation control processing unit **508**, a tessellation evaluation processing unit **512**, a geometry processing unit **516**, and a fragment/pixel processing unit **524**. The functions of data assembler **502**, primitive assemblers **506**, **514**, **518**, tessellation unit **510**, rasterizer **522**, and raster operations unit **526** may also be performed by other processing engines within a processing cluster (e.g., processing cluster **214** of FIG. **2**A) and a corresponding partition unit (e.g., partition unit **220**A-**220**N of FIG. **2**A). The graphics processing pipeline **500** may also be implemented using dedicated processing units for one or more functions. It is also possible that one or more portions of the graphics processing pipeline **500** are performed by parallel processing logic within a general-purpose processor (e.g., CPU). Optionally, one or more portions of the graphics processing pipeline **500** can access on-chip memory (e.g., parallel processor memory **222** as in FIG. **2**A) via a memory interface **528**, which may be an instance of the memory interface **218** of FIG. **2**A. The graphics processor pipeline **500**

may also be implemented via a multi-core group **365**A as in FIG. **3**C.

(163) The data assembler **502** is a processing unit that may collect vertex data for surfaces and primitives. The data assembler **502** then outputs the vertex data, including the vertex attributes, to the vertex processing unit **504**. The vertex processing unit **504** is a programmable execution unit that executes vertex shader programs, lighting and transforming vertex data as specified by the vertex shader programs. The vertex processing unit **504** reads data that is stored in cache, local or system memory for use in processing the vertex data and may be programmed to transform the vertex data from an object-based coordinate representation to a world space coordinate space or a normalized device coordinate space.

(164) A first instance of a primitive assembler **506** receives vertex attributes from the vertex processing unit **504**. The primitive assembler **506** readings stored vertex attributes as needed and constructs graphics primitives for processing by tessellation control processing unit **508**. The graphics primitives include triangles, line segments, points, patches, and so forth, as supported by various graphics processing application programming interfaces (APIs).

(165) The tessellation control processing unit **508** treats the input vertices as control points for a geometric patch. The control points are transformed from an input representation from the patch (e.g., the patch's bases) to a representation that is suitable for use in surface evaluation by the tessellation evaluation processing unit **512**. The tessellation control processing unit **508** can also compute tessellation factors for edges of geometric patches. A tessellation factor applies to a single edge and quantifies a view-dependent level of detail associated with the edge. A tessellation unit **510** is configured to receive the tessellation factors for edges of a patch and to tessellate the patch into multiple geometric primitives such as line, triangle, or quadrilateral primitives, which are transmitted to a tessellation evaluation processing unit **512**. The tessellation evaluation processing unit **512** operates on parameterized coordinates of the subdivided patch to generate a surface representation and vertex attributes for each vertex associated with the geometric primitives.

(166) A second instance of a primitive assembler **514** receives vertex attributes from the tessellation evaluation processing unit **512**, reading stored vertex attributes as needed, and constructs graphics primitives for processing by the geometry processing unit **516**. The geometry processing unit **516** is a programmable execution unit that executes geometry shader programs to transform graphics primitives received from primitive assembler **514** as specified by the geometry shader programs. The geometry processing unit **516** may be programmed to subdivide the graphics primitives into one or more new graphics primitives and calculate parameters used to rasterize the new graphics primitives.

(167) The geometry processing unit **516** may be able to add or delete elements in the geometry stream. The geometry processing unit **516** outputs the parameters and vertices specifying new graphics primitives to primitive assembler **518**. The primitive assembler **518** receives the parameters and vertices from the geometry processing unit **516** and constructs graphics primitives for processing by a viewport scale, cull, and clip unit **520**. The geometry processing unit **516** reads data that is stored in parallel processor memory or system memory for use in processing the geometry data. The viewport scale, cull, and clip unit **520** performs clipping, culling, and viewport scaling and outputs processed graphics primitives to a rasterizer **522**.

(168) The rasterizer **522** can perform depth culling and other depth-based optimizations. The rasterizer **522** also performs scan conversion on the new graphics primitives to generate fragments and output those fragments and associated coverage data to the fragment/pixel processing unit **524**. The fragment/pixel processing unit **524** is a programmable execution unit that is configured to execute fragment shader programs or pixel shader programs. The fragment/pixel processing unit **524** transforming fragments or pixels received from rasterizer **522**, as specified by the fragment or pixel shader programs. For example, the fragment/pixel processing unit **524** may be programmed to perform operations included but not limited to texture mapping, shading, blending, texture correction and perspective correction to produce shaded fragments or pixels that are output to a

raster operations unit **526**. The fragment/pixel processing unit **524** can read data that is stored in either the parallel processor memory or the system memory for use when processing the fragment data. Fragment or pixel shader programs may be configured to shade at sample, pixel, tile, or other granularities depending on the sampling rate configured for the processing units.

(169) The raster operations unit **526** is a processing unit that performs raster operations including, but not limited to stencil, z-test, blending, and the like, and outputs pixel data as processed graphics data to be stored in graphics memory (e.g., parallel processor memory **222** as in FIG. **2**A, and/or system memory **104** as in FIG. **1**), to be displayed on the one or more display device(s) **110**A-**110**B or for further processing by one of the one or more processor(s) **102** or parallel processor(s) **112**. The raster operations unit **526** may be configured to compress z or color data that is written to memory and decompress z or color data that is read from memory.

(170) Machine Learning Overview

(171) The architecture described above can be applied to perform training and inference operations using machine learning models. Machine learning has been successful at solving many kinds of tasks. The computations that arise when training and using machine learning algorithms (e.g., neural networks) lend themselves naturally to efficient parallel implementations. Accordingly, parallel processors such as general-purpose graphic processing units (GPGPUs) have played a significant role in the practical implementation of deep neural networks. Parallel graphics processors with single instruction, multiple thread (SIMT) architectures are designed to maximize the amount of parallel processing in the graphics pipeline. In an SIMT architecture, groups of parallel threads attempt to execute program instructions synchronously together as often as possible to increase processing efficiency. The efficiency provided by parallel machine learning algorithm implementations allows the use of high capacity networks and enables those networks to be trained on larger datasets.

(172) A machine learning algorithm is an algorithm that can learn based on a set of data. For example, machine learning algorithms can be designed to model high-level abstractions within a data set. For example, image recognition algorithms can be used to determine which of several categories to which a given input belong; regression algorithms can output a numerical value given an input; and pattern recognition algorithms can be used to generate translated text or perform text to speech and/or speech recognition.

(173) An exemplary type of machine learning algorithm is a neural network. There are many types of neural networks; a simple type of neural network is a feedforward network. A feedforward network may be implemented as an acyclic graph in which the nodes are arranged in layers. Typically, a feedforward network topology includes an input layer and an output layer that are separated by at least one hidden layer. The hidden layer transforms input received by the input layer into a representation that is useful for generating output in the output layer. The network nodes are fully connected via edges to the nodes in adjacent layers, but there are no edges between nodes within each layer. Data received at the nodes of an input layer of a feedforward network are propagated (i.e., “fed forward”) to the nodes of the output layer via an activation function that calculates the states of the nodes of each successive layer in the network based on coefficients (“weights”) respectively associated with each of the edges connecting the layers. Depending on the specific model being represented by the algorithm being executed, the output from the neural network algorithm can take various forms.

(174) Before a machine learning algorithm can be used to model a particular problem, the algorithm is trained using a training data set. Training a neural network involves selecting a network topology, using a set of training data representing a problem being modeled by the network, and adjusting the weights until the network model performs with a minimal error for all instances of the training data set. For example, during a supervised learning training process for a neural network, the output produced by the network in response to the input representing an instance in a training data set is compared to the “correct” labeled output for that instance, an error

signal representing the difference between the output and the labeled output is calculated, and the weights associated with the connections are adjusted to minimize that error as the error signal is backward propagated through the layers of the network. The network is considered "trained" when the errors for each of the outputs generated from the instances of the training data set are minimized.

(175) The accuracy of a machine learning algorithm can be affected significantly by the quality of the data set used to train the algorithm. The training process can be computationally intensive and may require a significant amount of time on a conventional general-purpose processor. Accordingly, parallel processing hardware is used to train many types of machine learning algorithms. This is particularly useful for optimizing the training of neural networks, as the computations performed in adjusting the coefficients in neural networks lend themselves naturally to parallel implementations. Specifically, many machine learning algorithms and software applications have been adapted to make use of the parallel processing hardware within general-purpose graphics processing devices.

(176) FIG. **6** is a generalized diagram of a machine learning software stack **600**. A machine learning application **602** can be configured to train a neural network using a training dataset or to use a trained deep neural network to implement machine intelligence. The machine learning application **602** can include training and inference functionality for a neural network and/or specialized software that can be used to train a neural network before deployment. The machine learning application **602** can implement any type of machine intelligence including but not limited to image recognition, mapping and localization, autonomous navigation, speech synthesis, medical imaging, or language translation.

(177) Hardware acceleration for the machine learning application **602** can be enabled via a machine learning framework **604**. The machine learning framework **604** can provide a library of machine learning primitives. Machine learning primitives are basic operations that are commonly performed by machine learning algorithms. Without the machine learning framework **604**, developers of machine learning algorithms would be required to create and optimize the main computational logic associated with the machine learning algorithm, then re-optimize the computational logic as new parallel processors are developed. Instead, the machine learning application can be configured to perform the necessary computations using the primitives provided by the machine learning framework **604**. Exemplary primitives include tensor convolutions, activation functions, and pooling, which are computational operations that are performed while training a convolutional neural network (CNN). The machine learning framework **604** can also provide primitives to implement basic linear algebra subprograms performed by many machine-learning algorithms, such as matrix and vector operations.

(178) The machine learning framework **604** can process input data received from the machine learning application **602** and generate the appropriate input to a compute framework **606**. The compute framework **606** can abstract the underlying instructions provided to the GPGPU driver **608** to enable the machine learning framework **604** to take advantage of hardware acceleration via the GPGPU hardware **610** without requiring the machine learning framework **604** to have intimate knowledge of the architecture of the GPGPU hardware **610**. Additionally, the compute framework **606** can enable hardware acceleration for the machine learning framework **604** across a variety of types and generations of the GPGPU hardware **610**.

(179) GPGPU Machine Learning Acceleration

(180) FIG. **7** illustrates a general-purpose graphics processing unit **700**, which may be the parallel processor **200** of FIG. **2**A or the parallel processor(s) **112** of FIG. **1**. The general-purpose processing unit (GPGPU) **700** may be configured to be particularly efficient in processing the type of computational workloads associated with training deep neural networks. Additionally, the GPGPU **700** can be linked directly to other instances of the GPGPU to create a multi-GPU cluster to improve training speed for particularly deep neural networks.

(181) The GPGPU **700** includes a host interface **702** to enable a connection with a host processor. The host interface **702** may be a PCI Express interface. However, the host interface can also be a vendor specific communications interface or communications fabric. The GPGPU **700** receives commands from the host processor and uses a global scheduler **704** to distribute execution threads associated with those commands to a set of processing clusters **706**A-**706**H. The processing clusters **706**A-**706**H share a cache memory **708**. The cache memory **708** can serve as a higher-level cache for cache memories within the processing clusters **706**A-**706**H. The illustrated processing clusters **706**A-**706**H may correspond with processing clusters **214**A-**214**N as in FIG. **2**A.
(182) The GPGPU **700** includes memory **714**A-**714**B coupled with the processing clusters **706**A-H via a set of memory controllers **712**A-**712**B. The memory **714**A-**714**B can include various types of memory devices including dynamic random-access memory (DRAM) or graphics random access memory, such as synchronous graphics random access memory (SGRAM), including graphics double data rate (GDDR) memory. The memory **714**A-**714**B may also include 3D stacked memory, including but not limited to high bandwidth memory (HBM).
(183) Each of the processing clusters **706**A-**706**H may include a set of graphics multiprocessors, such as the graphics multiprocessor **234** of FIG. **2**D, graphics multiprocessor **325** of FIG. **3**A, graphics multiprocessor **350** of FIG. **3**B, or may include a multi-core group **365**A-**365**N as in FIG. **3**C. The graphics multiprocessors of the compute cluster include multiple types of integer and floating-point logic units that can perform computational operations at a range of precisions including suited for machine learning computations. For example, at least a subset of the floating-point units in each of the processing clusters **706**A-**706**H can be configured to perform 16-bit or 32-bit floating point operations, while a different subset of the floating-point units can be configured to perform 64-bit floating point operations.
(184) Multiple instances of the GPGPU **700** can be configured to operate as a compute cluster. The communication mechanism used by the compute cluster for synchronization and data exchange varies across embodiments. For example, the multiple instances of the GPGPU **700** communicate over the host interface **702**. In one embodiment the GPGPU **700** includes an I/O hub **709** that couples the GPGPU **700** with a GPU link **710** that enables a direct connection to other instances of the GPGPU. The GPU link **710** may be coupled to a dedicated GPU-to-GPU bridge that enables communication and synchronization between multiple instances of the GPGPU **700**. Optionally, the GPU link **710** couples with a high-speed interconnect to transmit and receive data to other GPGPUs or parallel processors. The multiple instances of the GPGPU **700** may be located in separate data processing systems and communicate via a network device that is accessible via the host interface **702**. The GPU link **710** may be configured to enable a connection to a host processor in addition to or as an alternative to the host interface **702**.
(185) While the illustrated configuration of the GPGPU **700** can be configured to train neural networks, an alternate configuration of the GPGPU **700** can be configured for deployment within a high performance or low power inferencing platform. In an inferencing configuration, the GPGPU **700** includes fewer of the processing clusters **706**A-**706**H relative to the training configuration. Additionally, memory technology associated with the memory **714**A-**714**B may differ between inferencing and training configurations. In one embodiment, the inferencing configuration of the GPGPU **700** can support inferencing specific instructions. For example, an inferencing configuration can provide support for one or more 8-bit integer dot product instructions, which are commonly used during inferencing operations for deployed neural networks.
(186) FIG. **8** illustrates a multi-GPU computing system **800**. The multi-GPU computing system **800** can include a processor **802** coupled to multiple GPGPUs **806**A-**806**D via a host interface switch **804**. The host interface switch **804** may be a PCI express switch device that couples the processor **802** to a PCI express bus over which the processor **802** can communicate with the set of GPGPUs **806**A-**806**D. Each of the multiple GPGPUs **806**A-**806**D can be an instance of the GPGPU **700** of FIG. **7**. The GPGPUs **806**A-**806**D can interconnect via a set of high-speed point to point GPU to

GPU links **816**. The high-speed GPU to GPU links can connect to each of the GPGPUs **806**A-**806**D via a dedicated GPU link, such as the GPU link **710** as in FIG. **7**. The P2P GPU links **816** enable direct communication between each of the GPGPUs **806**A-**806**D without requiring communication over the host interface bus to which the processor **802** is connected. With GPU-to-GPU traffic directed to the P2P GPU links, the host interface bus remains available for system memory access or to communicate with other instances of the multi-GPU computing system **800**, for example, via one or more network devices. While in FIG. **8** the GPGPUs **806**A-**806**D connect to the processor **802** via the host interface switch **804**, the processor **802** may alternatively include direct support for the P2P GPU links **816** and connect directly to the GPGPUs **806**A-**806**D.

(187) Machine Learning Neural Network Implementations

(188) The computing architecture described herein can be configured to perform the types of parallel processing that is particularly suited for training and deploying neural networks for machine learning. A neural network can be generalized as a network of functions having a graph relationship. As is well-known in the art, there are a variety of types of neural network implementations used in machine learning. One exemplary type of neural network is the feedforward network, as previously described.

(189) A second exemplary type of neural network is the Convolutional Neural Network (CNN). A CNN is a specialized feedforward neural network for processing data having a known, grid-like topology, such as image data. Accordingly, CNNs are commonly used for compute vision and image recognition applications, but they also may be used for other types of pattern recognition such as speech and language processing. The nodes in the CNN input layer are organized into a set of “filters” (feature detectors inspired by the receptive fields found in the retina), and the output of each set of filters is propagated to nodes in successive layers of the network. The computations for a CNN include applying the convolution mathematical operation to each filter to produce the output of that filter. Convolution is a specialized kind of mathematical operation performed by two functions to produce a third function that is a modified version of one of the two original functions. In convolutional network terminology, the first function to the convolution can be referred to as the input, while the second function can be referred to as the convolution kernel. The output may be referred to as the feature map. For example, the input to a convolutional layer can be a multidimensional array of data that defines the various color components of an input image. The convolution kernel can be a multidimensional array of parameters, where the parameters are adapted by the training process for the neural network.

(190) Recurrent neural networks (RNNs) are a family of feedforward neural networks that include feedback connections between layers. RNNs enable modeling of sequential data by sharing parameter data across different parts of the neural network. The architecture for an RNN includes cycles. The cycles represent the influence of a present value of a variable on its own value at a future time, as at least a portion of the output data from the RNN is used as feedback for processing subsequent input in a sequence. This feature makes RNNs particularly useful for language processing due to the variable nature in which language data can be composed.

(191) The figures described below present exemplary feedforward, CNN, and RNN networks, as well as describe a general process for respectively training and deploying each of those types of networks. It will be understood that these descriptions are exemplary and non-limiting as to any specific embodiment described herein and the concepts illustrated can be applied generally to deep neural networks and machine learning techniques in general.

(192) The exemplary neural networks described above can be used to perform deep learning. Deep learning is machine learning using deep neural networks. The deep neural networks used in deep learning are artificial neural networks composed of multiple hidden layers, as opposed to shallow neural networks that include only a single hidden layer. Deeper neural networks are generally more computationally intensive to train. However, the additional hidden layers of the network enable multistep pattern recognition that results in reduced output error relative to shallow machine

learning techniques.
(193) Deep neural networks used in deep learning typically include a front-end network to perform feature recognition coupled to a back-end network which represents a mathematical model that can perform operations (e.g., object classification, speech recognition, etc.) based on the feature representation provided to the model. Deep learning enables machine learning to be performed without requiring hand crafted feature engineering to be performed for the model. Instead, deep neural networks can learn features based on statistical structure or correlation within the input data. The learned features can be provided to a mathematical model that can map detected features to an output. The mathematical model used by the network is generally specialized for the specific task to be performed, and different models will be used to perform different task.
(194) Once the neural network is structured, a learning model can be applied to the network to train the network to perform specific tasks. The learning model describes how to adjust the weights within the model to reduce the output error of the network. Backpropagation of errors is a common method used to train neural networks. An input vector is presented to the network for processing. The output of the network is compared to the desired output using a loss function and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards until each neuron has an associated error value which roughly represents its contribution to the original output. The network can then learn from those errors using an algorithm, such as the stochastic gradient descent algorithm, to update the weights of the of the neural network.
(195) FIG. **9**A-**9**B illustrate an exemplary convolutional neural network. FIG. **9**A illustrates various layers within a CNN. As shown in FIG. **9**A, an exemplary CNN used to model image processing can receive input **902** describing the red, green, and blue (RGB) components of an input image. The input **902** can be processed by multiple convolutional layers (e.g., convolutional layer **904**, convolutional layer **906**). The output from the multiple convolutional layers may optionally be processed by a set of fully connected layers **908**. Neurons in a fully connected layer have full connections to all activations in the previous layer, as previously described for a feedforward network. The output from the fully connected layers **908** can be used to generate an output result from the network. The activations within the fully connected layers **908** can be computed using matrix multiplication instead of convolution. Not all CNN implementations make use of fully connected layers **908**. For example, in some implementations the convolutional layer **906** can generate output for the CNN.
(196) The convolutional layers are sparsely connected, which differs from traditional neural network configuration found in the fully connected layers **908**. Traditional neural network layers are fully connected, such that every output unit interacts with every input unit. However, the convolutional layers are sparsely connected because the output of the convolution of a field is input (instead of the respective state value of each of the nodes in the field) to the nodes of the subsequent layer, as illustrated. The kernels associated with the convolutional layers perform convolution operations, the output of which is sent to the next layer. The dimensionality reduction performed within the convolutional layers is one aspect that enables the CNN to scale to process large images.
(197) FIG. **9**B illustrates exemplary computation stages within a convolutional layer of a CNN. Input to a convolutional layer **912** of a CNN can be processed in three stages of a convolutional layer **914**. The three stages can include a convolution stage **916**, a detector stage **918**, and a pooling stage **920**. The convolutional layer **914** can then output data to a successive convolutional layer. The final convolutional layer of the network can generate output feature map data or provide input to a fully connected layer, for example, to generate a classification value for the input to the CNN.
(198) In the convolution stage **916** performs several convolutions in parallel to produce a set of linear activations. The convolution stage **916** can include an affine transformation, which is any transformation that can be specified as a linear transformation plus a translation. Affine transformations include rotations, translations, scaling, and combinations of these transformations.

The convolution stage computes the output of functions (e.g., neurons) that are connected to specific regions in the input, which can be determined as the local region associated with the neuron. The neurons compute a dot product between the weights of the neurons and the region in the local input to which the neurons are connected. The output from the convolution stage **916** defines a set of linear activations that are processed by successive stages of the convolutional layer **914**.

(199) The linear activations can be processed by a detector stage **918**. In the detector stage **918**, each linear activation is processed by a non-linear activation function. The non-linear activation function increases the nonlinear properties of the overall network without affecting the receptive fields of the convolutional layer. Several types of non-linear activation functions may be used. One particular type is the rectified linear unit (ReLU), which uses an activation function defined as $f(x)=\max(0, x)$, such that the activation is thresholded at zero.

(200) The pooling stage **920** uses a pooling function that replaces the output of the convolutional layer **906** with a summary statistic of the nearby outputs. The pooling function can be used to introduce translation invariance into the neural network, such that small translations to the input do not change the pooled outputs. Invariance to local translation can be useful in scenarios where the presence of a feature in the input data is more important than the precise location of the feature. Various types of pooling functions can be used during the pooling stage **920**, including max pooling, average pooling, and l2-norm pooling. Additionally, some CNN implementations do not include a pooling stage. Instead, such implementations substitute and additional convolution stage having an increased stride relative to previous convolution stages.

(201) The output from the convolutional layer **914** can then be processed by the next layer **922**. The next layer **922** can be an additional convolutional layer or one of the fully connected layers **908**. For example, the first convolutional layer **904** of FIG. **9**A can output to the second convolutional layer **906**, while the second convolutional layer can output to a first layer of the fully connected layers **908**.

(202) FIG. **10** illustrates an exemplary recurrent neural network **1000**. In a recurrent neural network (RNN), the previous state of the network influences the output of the current state of the network. RNNs can be built in a variety of ways using a variety of functions. The use of RNNs generally revolves around using mathematical models to predict the future based on a prior sequence of inputs. For example, an RNN may be used to perform statistical language modeling to predict an upcoming word given a previous sequence of words. The illustrated RNN **1000** can be described has having an input layer **1002** that receives an input vector, hidden layers **1004** to implement a recurrent function, a feedback mechanism **1005** to enable a ‘memory’ of previous states, and an output layer **1006** to output a result. The RNN **1000** operates based on time-steps. The state of the RNN at a given time step is influenced based on the previous time step via the feedback mechanism **1005**. For a given time step, the state of the hidden layers **1004** is defined by the previous state and the input at the current time step. An initial input (x_1) at a first time step can be processed by the hidden layer **1004**. A second input (x_2) can be processed by the hidden layer **1004** using state information that is determined during the processing of the initial input (x_1). A given state can be computed as $s_t=f(Ux_t+Ws_{t-1})$, where U and W are parameter matrices. The function f is generally a nonlinearity, such as the hyperbolic tangent function (Tanh) or a variant of the rectifier function $f(x)=\max(0, x)$. However, the specific mathematical function used in the hidden layers **1004** can vary depending on the specific implementation details of the RNN **1000**.

(203) In addition to the basic CNN and RNN networks described, variations on those networks may be enabled. One example RNN variant is the long short term memory (LSTM) RNN. LSTM RNNs are capable of learning long-term dependencies that may be necessary for processing longer sequences of language. A variant on the CNN is a convolutional deep belief network, which has a structure similar to a CNN and is trained in a manner similar to a deep belief network. A deep belief

network (DBN) is a generative neural network that is composed of multiple layers of stochastic (random) variables. DBNs can be trained layer-by-layer using greedy unsupervised learning. The learned weights of the DBN can then be used to provide pre-train neural networks by determining an optimal initial set of weights for the neural network.

(204) FIG. **11** illustrates training and deployment of a deep neural network. Once a given network has been structured for a task the neural network is trained using a training dataset **1102**. Various training frameworks **1104** have been developed to enable hardware acceleration of the training process. For example, the machine learning framework **604** of FIG. **6** may be configured as a training framework **1104**. The training framework **1104** can hook into an untrained neural network **1106** and enable the untrained neural net to be trained using the parallel processing resources described herein to generate a trained neural network **1108**.

(205) To start the training process the initial weights may be chosen randomly or by pre-training using a deep belief network. The training cycle then be performed in either a supervised or unsupervised manner.

(206) Supervised learning is a learning method in which training is performed as a mediated operation, such as when the training dataset **1102** includes input paired with the desired output for the input, or where the training dataset includes input having known output and the output of the neural network is manually graded. The network processes the inputs and compares the resulting outputs against a set of expected or desired outputs. Errors are then propagated back through the system. The training framework **1104** can adjust to adjust the weights that control the untrained neural network **1106**. The training framework **1104** can provide tools to monitor how well the untrained neural network **1106** is converging towards a model suitable to generating correct answers based on known input data. The training process occurs repeatedly as the weights of the network are adjusted to refine the output generated by the neural network. The training process can continue until the neural network reaches a statistically desired accuracy associated with a trained neural network **1108**. The trained neural network **1108** can then be deployed to implement any number of machine learning operations to generate an inference result **1114** based on input of new data **1112**.

(207) Unsupervised learning is a learning method in which the network attempts to train itself using unlabeled data. Thus, for unsupervised learning the training dataset **1102** will include input data without any associated output data. The untrained neural network **1106** can learn groupings within the unlabeled input and can determine how individual inputs are related to the overall dataset. Unsupervised training can be used to generate a self-organizing map, which is a type of trained neural network **1108** capable of performing operations useful in reducing the dimensionality of data. Unsupervised training can also be used to perform anomaly detection, which allows the identification of data points in an input dataset that deviate from the normal patterns of the data.

(208) Variations on supervised and unsupervised training may also be employed. Semi-supervised learning is a technique in which in the training dataset **1102** includes a mix of labeled and unlabeled data of the same distribution. Incremental learning is a variant of supervised learning in which input data is continuously used to further train the model. Incremental learning enables the trained neural network **1108** to adapt to the new data **1112** without forgetting the knowledge instilled within the network during initial training.

(209) Whether supervised or unsupervised, the training process for particularly deep neural networks may be too computationally intensive for a single compute node. Instead of using a single compute node, a distributed network of computational nodes can be used to accelerate the training process.

(210) FIG. **12** is a block diagram illustrating distributed learning. Distributed learning is a training model that uses multiple distributed computing nodes to perform supervised or unsupervised training of a neural network. The distributed computational nodes can each include one or more host processors and one or more of the general-purpose processing nodes, such as the highly

parallel general-purpose graphics processing unit **700** as in FIG. **7**. As illustrated, distributed learning can be performed model parallelism **1202**, data parallelism **1204**, or a combination of model and data parallelism **1204**.
(211) In model parallelism **1202**, different computational nodes in a distributed system can perform training computations for different parts of a single network. For example, each layer of a neural network can be trained by a different processing node of the distributed system. The benefits of model parallelism include the ability to scale to particularly large models. Splitting the computations associated with different layers of the neural network enables the training of very large neural networks in which the weights of all layers would not fit into the memory of a single computational node. In some instances, model parallelism can be particularly useful in performing unsupervised training of large neural networks.
(212) In data parallelism **1204**, the different nodes of the distributed network have a complete instance of the model and each node receives a different portion of the data. The results from the different nodes are then combined. While different approaches to data parallelism are possible, data parallel training approaches all require a technique of combining results and synchronizing the model parameters between each node. Exemplary approaches to combining data include parameter averaging and update based data parallelism. Parameter averaging trains each node on a subset of the training data and sets the global parameters (e.g., weights, biases) to the average of the parameters from each node. Parameter averaging uses a central parameter server that maintains the parameter data. Update based data parallelism is similar to parameter averaging except that instead of transferring parameters from the nodes to the parameter server, the updates to the model are transferred. Additionally, update based data parallelism can be performed in a decentralized manner, where the updates are compressed and transferred between nodes.
(213) Combined model and data parallelism **1206** can be implemented, for example, in a distributed system in which each computational node includes multiple GPUs. Each node can have a complete instance of the model with separate GPUs within each node are used to train different portions of the model.
(214) Distributed training has increased overhead relative to training on a single machine. However, the parallel processors and GPGPUs described herein can each implement various techniques to reduce the overhead of distributed training, including techniques to enable high bandwidth GPU-to-GPU data transfer and accelerated remote data synchronization.
(215) Exemplary Machine Learning Applications
(216) Machine learning can be applied to solve a variety of technological problems, including but not limited to computer vision, autonomous driving and navigation, speech recognition, and language processing. Computer vision has traditionally been one of the most active research areas for machine learning applications. Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating new categories of visual abilities. For example, computer vision applications can be configured to recognize sound waves from the vibrations induced in objects visible in a video. Parallel processor accelerated machine learning enables computer vision applications to be trained using significantly larger training dataset than previously feasible and enables inferencing systems to be deployed using low power parallel processors.
(217) Parallel processor accelerated machine learning has autonomous driving applications including lane and road sign recognition, obstacle avoidance, navigation, and driving control. Accelerated machine learning techniques can be used to train driving models based on datasets that define the appropriate responses to specific training input. The parallel processors described herein can enable rapid training of the increasingly complex neural networks used for autonomous driving solutions and enables the deployment of low power inferencing processors in a mobile platform suitable for integration into autonomous vehicles.
(218) Parallel processor accelerated deep neural networks have enabled machine learning

approaches to automatic speech recognition (ASR). ASR includes the creation of a function that computes the most probable linguistic sequence given an input acoustic sequence. Accelerated machine learning using deep neural networks have enabled the replacement of the hidden Markov models (HMMs) and Gaussian mixture models (GMMs) previously used for ASR.
(219) Parallel processor accelerated machine learning can also be used to accelerate natural language processing. Automatic learning procedures can make use of statistical inference algorithms to produce models that are robust to erroneous or unfamiliar input. Exemplary natural language processor applications include automatic machine translation between human languages.
(220) The parallel processing platforms used for machine learning can be divided into training platforms and deployment platforms. Training platforms are generally highly parallel and include optimizations to accelerate multi-GPU single node training and multi-node, multi-GPU training. Exemplary parallel processors suited for training include the general-purpose graphics processing unit **700** of FIG. **7** and the multi-GPU computing system **800** of FIG. **8**. On the contrary, deployed machine learning platforms generally include lower power parallel processors suitable for use in products such as cameras, autonomous robots, and autonomous vehicles.
(221) FIG. **13** illustrates an exemplary inferencing system on a chip (SOC) **1300** suitable for performing inferencing using a trained model. The SOC **1300** can integrate processing components including a media processor **1302**, a vision processor **1304**, a GPGPU **1306** and a multi-core processor **1308**. The GPGPU **1306** may be a GPGPU as described herein, such as the GPGPU **700**, and the multi-core processor **1308** may be a multi-core processor described herein, such as the multi-core processors **405-406**. The SOC **1300** can additionally include on-chip memory **1305** that can enable a shared on-chip data pool that is accessible by each of the processing components. The processing components can be optimized for low power operation to enable deployment to a variety of machine learning platforms, including autonomous vehicles and autonomous robots. For example, one implementation of the SOC **1300** can be used as a portion of the main control system for an autonomous vehicle. Where the SOC **1300** is configured for use in autonomous vehicles the SOC is designed and configured for compliance with the relevant functional safety standards of the deployment jurisdiction.
(222) During operation, the media processor **1302** and vision processor **1304** can work in concert to accelerate computer vision operations. The media processor **1302** can enable low latency decode of multiple high-resolution (e.g., 4K, 8K) video streams. The decoded video streams can be written to a buffer in the on-chip memory **1305**. The vision processor **1304** can then parse the decoded video and perform preliminary processing operations on the frames of the decoded video in preparation of processing the frames using a trained image recognition model. For example, the vision processor **1304** can accelerate convolution operations for a CNN that is used to perform image recognition on the high-resolution video data, while back end model computations are performed by the GPGPU **1306**.
(223) The multi-core processor **1308** can include control logic to assist with sequencing and synchronization of data transfers and shared memory operations performed by the media processor **1302** and the vision processor **1304**. The multi-core processor **1308** can also function as an application processor to execute software applications that can make use of the inferencing compute capability of the GPGPU **1306**. For example, at least a portion of the navigation and driving logic can be implemented in software executing on the multi-core processor **1308**. Such software can directly issue computational workloads to the GPGPU **1306** or the computational workloads can be issued to the multi-core processor **1308**, which can offload at least a portion of those operations to the GPGPU **1306**.
(224) The GPGPU **1306** can include compute clusters such as a low power configuration of the processing clusters **706**A-**706**H within general-purpose graphics processing unit **700**. The compute clusters within the GPGPU **1306** can support instruction that are specifically optimized to perform inferencing computations on a trained neural network. For example, the GPGPU **1306** can support

instructions to perform low precision computations such as 8-bit and 4-bit integer vector operations.

(225) Additional System Overview

(226) FIG. **14** is a block diagram of a processing system **1400**. The elements of FIG. **14** having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. System **1400** may be used in a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors **1402** or processor cores **1407**. The system **1400** may be a processing platform incorporated within a system-on-a-chip (SoC) integrated circuit for use in mobile, handheld, or embedded devices such as within Internet-of-things (IoT) devices with wired or wireless connectivity to a local or wide area network.

(227) The system **1400** may be a processing system having components that correspond with those of FIG. **1**. For example, in different configurations, processor(s) **1402** or processor core(s) **1407** may correspond with processor(s) **102** of FIG. **1**. Graphics processor(s) **1408** may correspond with parallel processor(s) **112** of FIG. **1**. External graphics processor **1418** may be one of the add-in device(s) **120** of FIG. **1**.

(228) The system **1400** can include, couple with, or be integrated within: a server-based gaming platform; a game console, including a game and media console; a mobile gaming console, a handheld game console, or an online game console. The system **1400** may be part of a mobile phone, smart phone, tablet computing device or mobile Internet-connected device such as a laptop with low internal storage capacity. Processing system **1400** can also include, couple with, or be integrated within: a wearable device, such as a smart watch wearable device; smart eyewear or clothing enhanced with augmented reality (AR) or virtual reality (VR) features to provide visual, audio or tactile outputs to supplement real world visual, audio or tactile experiences or otherwise provide text, audio, graphics, video, holographic images or video, or tactile feedback; other augmented reality (AR) device; or other virtual reality (VR) device. The processing system **1400** may include or be part of a television or set top box device. The system **1400** can include, couple with, or be integrated within a self-driving vehicle such as a bus, tractor trailer, car, motor or electric power cycle, plane or glider (or any combination thereof). The self-driving vehicle may use system **1400** to process the environment sensed around the vehicle.

(229) The one or more processors **1402** may include one or more processor cores **1407** to process instructions which, when executed, perform operations for system or user software. The least one of the one or more processor cores **1407** may be configured to process a specific instruction set **1409**. The instruction set **1409** may facilitate Complex Instruction Set Computing (CISC), Reduced Instruction Set Computing (RISC), or computing via a Very Long Instruction Word (VLIW). One or more processor cores **1407** may process a different instruction set **1409**, which may include instructions to facilitate the emulation of other instruction sets. Processor core **1407** may also include other processing devices, such as a Digital Signal Processor (DSP).

(230) The processor **1402** may include cache memory **1404**. Depending on the architecture, the processor **1402** can have a single internal cache or multiple levels of internal cache. In some embodiments, the cache memory is shared among various components of the processor **1402**. In some embodiments, the processor **1402** also uses an external cache (e.g., a Level-3 (L3) cache or Last Level Cache (LLC)) (not shown), which may be shared among processor cores **1407** using known cache coherency techniques. A register file **1406** can be additionally included in processor **1402** and may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor **1402**.

(231) The one or more processor(s) **1402** may be coupled with one or more interface bus(es) **1410**

to transmit communication signals such as address, data, or control signals between processor **1402** and other components in the system **1400**. The interface bus **1410**, in one of these embodiments, can be a processor bus, such as a version of the Direct Media Interface (DMI) bus. However, processor busses are not limited to the DMI bus, and may include one or more Peripheral Component Interconnect buses (e.g., PCI, PCI express), memory busses, or other types of interface busses. For example, the processor(s) **1402** may include an integrated memory controller **1416** and a platform controller hub **1430**. The memory controller **1416** facilitates communication between a memory device and other components of the system **1400**, while the platform controller hub (PCH) **1430** provides connections to I/O devices via a local I/O bus.

(232) The memory device **1420** can be a dynamic random-access memory (DRAM) device, a static random-access memory (SRAM) device, flash memory device, phase-change memory device, or some other memory device having suitable performance to serve as process memory. The memory device **1420** can, for example, operate as system memory for the system **1400**, to store data **1422** and instructions **1421** for use when the one or more processors **1402** executes an application or process. Memory controller **1416** also couples with an optional external graphics processor **1418**, which may communicate with the one or more graphics processors **1408** in processors **1402** to perform graphics and media operations. In some embodiments, graphics, media, and or compute operations may be assisted by an accelerator **1412** which is a coprocessor that can be configured to perform a specialized set of graphics, media, or compute operations. For example, the accelerator **1412** may be a matrix multiplication accelerator used to optimize machine learning or compute operations. The accelerator **1412** can be a ray-tracing accelerator that can be used to perform ray-tracing operations in concert with the graphics processor **1408**. In one embodiment, an external accelerator **1419** may be used in place of or in concert with the accelerator **1412**.

(233) A display device **1411** may be provided that can connect to the processor(s) **1402**. The display device **1411** can be one or more of an internal display device, as in a mobile electronic device or a laptop device or an external display device attached via a display interface (e.g., DisplayPort, etc.). The display device **1411** can be a head mounted display (HMD) such as a stereoscopic display device for use in virtual reality (VR) applications or augmented reality (AR) applications.

(234) The platform controller hub **1430** may enable peripherals to connect to memory device **1420** and processor **1402** via a high-speed I/O bus. The I/O peripherals include, but are not limited to, an audio controller **1446**, a network controller **1434**, a firmware interface **1428**, a wireless transceiver **1426**, touch sensors **1425**, a data storage device **1424** (e.g., non-volatile memory, volatile memory, hard disk drive, flash memory, NAND, 3D NAND, 3D XPoint/Optane, etc.). The data storage device **1424** can connect via a storage interface (e.g., SATA) or via a peripheral bus, such as a Peripheral Component Interconnect bus (e.g., PCI, PCI express). The touch sensors **1425** can include touch screen sensors, pressure sensors, or fingerprint sensors. The wireless transceiver **1426** can be a Wi-Fi transceiver, a Bluetooth transceiver, or a mobile network transceiver such as a 3G, 4G, 5G, or Long-Term Evolution (LTE) transceiver. The firmware interface **1428** enables communication with system firmware, and can be, for example, a unified extensible firmware interface (UEFI). The network controller **1434** can enable a network connection to a wired network. In some embodiments, a high-performance network controller (not shown) couples with the interface bus **1410**. The audio controller **1446** may be a multi-channel high definition audio controller. In some of these embodiments the system **1400** includes an optional legacy I/O controller **1440** for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. The platform controller hub **1430** can also connect to one or more Universal Serial Bus (USB) controllers **1442** connect input devices, such as keyboard and mouse **1443** combinations, a camera **1444**, or other USB input devices.

(235) It will be appreciated that the system **1400** shown is exemplary and not limiting, as other types of data processing systems that are differently configured may also be used. For example, an

instance of the memory controller **1416** and platform controller hub **1430** may be integrated into a discreet external graphics processor, such as the external graphics processor **1418**. The platform controller hub **1430** and/or memory controller **1416** may be external to the one or more processor(s) **1402**. For example, the system **1400** can include an external memory controller **1416** and platform controller hub **1430**, which may be configured as a memory controller hub and peripheral controller hub within a system chipset that is in communication with the processor(s) **1402**.

(236) For example, circuit boards ("sleds") can be used on which components such as CPUs, memory, and other components are placed are designed for increased thermal performance. Processing components such as the processors may be located on a top side of a sled while near memory, such as DIMMs, are located on a bottom side of the sled. As a result of the enhanced airflow provided by this design, the components may operate at higher frequencies and power levels than in typical systems, thereby increasing performance. Furthermore, the sleds are configured to blindly mate with power and data communication cables in a rack, thereby enhancing their ability to be quickly removed, upgraded, reinstalled, and/or replaced. Similarly, individual components located on the sleds, such as processors, accelerators, memory, and data storage drives, are configured to be easily upgraded due to their increased spacing from each other. In the illustrative embodiment, the components additionally include hardware attestation features to prove their authenticity.

(237) A data center can utilize a single network architecture ("fabric") that supports multiple other network architectures including Ethernet and Omni-Path. The sleds can be coupled to switches via optical fibers, which provide higher bandwidth and lower latency than typical twisted pair cabling (e.g., Category 5, Category 5e, Category 6, etc.). Due to the high bandwidth, low latency interconnections and network architecture, the data center may, in use, pool resources, such as memory, accelerators (e.g., GPUs, graphics accelerators, FPGAs, ASICs, neural network and/or artificial intelligence accelerators, etc.), and data storage drives that are physically disaggregated, and provide them to compute resources (e.g., processors) on an as needed basis, enabling the compute resources to access the pooled resources as if they were local.

(238) A power supply or source can provide voltage and/or current to system **1400** or any component or system described herein. In one example, the power supply includes an AC to DC (alternating current to direct current) adapter to plug into a wall outlet. Such AC power can be renewable energy (e.g., solar power) power source. In one example, the power source includes a DC power source, such as an external AC to DC converter. A power source or power supply may also include wireless charging hardware to charge via proximity to a charging field. The power source can include an internal battery, alternating current supply, motion-based power supply, solar power supply, or fuel cell source.

(239) FIG. **15**A-**15**C illustrate computing systems and graphics processors. The elements of FIG. **15**A-**15**C having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

(240) FIG. **15**A is a block diagram of a processor **1500**, which may be a variant of one of the processors **1402** and may be used in place of one of those. Therefore, the disclosure of any features in combination with the processor **1500** herein also discloses a corresponding combination with the processor(s) **1402**, but is not limited to such. The processor **1500** may have one or more processor cores **1502**A-**1502**N, an integrated memory controller **1514**, and an integrated graphics processor **1508**. Where an integrated graphics processor **1508** is excluded, the system that includes the processor will include a graphics processor device within a system chipset or coupled via a system bus. Processor **1500** can include additional cores up to and including additional core **1502**N represented by the dashed lined boxes. Each of processor cores **1502**A-**1502**N includes one or more

internal cache units **1504**A-**1504**N. In some embodiments each processor core **1502**A-**1502**N also has access to one or more shared cache units **1506**. The internal cache units **1504**A-**1504**N and shared cache units **1506** represent a cache memory hierarchy within the processor **1500**. The cache memory hierarchy may include at least one level of instruction and data cache within each processor core and one or more levels of shared mid-level cache, such as a Level 2 (L2), Level 3 (L3), Level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the LLC. In some embodiments, cache coherency logic maintains coherency between the various cache units **1506** and **1504**A-**1504**N.
(241) The processor **1500** may also include a set of one or more bus controller units **1516** and a system agent core **1510**. The one or more bus controller units **1516** manage a set of peripheral buses, such as one or more PCI or PCI express busses. System agent core **1510** provides management functionality for the various processor components. The system agent core **1510** may include one or more integrated memory controllers **1514** to manage access to various external memory devices (not shown).
(242) For example, one or more of the processor cores **1502**A-**1502**N may include support for simultaneous multi-threading. The system agent core **1510** includes components for coordinating and operating cores **1502**A-**1502**N during multi-threaded processing. System agent core **1510** may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of processor cores **1502**A-**1502**N and graphics processor **1508**.
(243) The processor **1500** may additionally include graphics processor **1508** to execute graphics processing operations. In some of these embodiments, the graphics processor **1508** couples with the set of shared cache units **1506**, and the system agent core **1510**, including the one or more integrated memory controllers **1514**. The system agent core **1510** may also include a display controller **1511** to drive graphics processor output to one or more coupled displays. The display controller **1511** may also be a separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor **1508**.
(244) A ring-based interconnect **1512** may be used to couple the internal components of the processor **1500**. However, an alternative interconnect unit may be used, such as a point-to-point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In some of these embodiments with a ring-based interconnect **1512**, the graphics processor **1508** couples with the ring-based interconnect **1512** via an I/O link **1513**.
(245) The exemplary I/O link **1513** represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module **1518**, such as an eDRAM module. Optionally, each of the processor cores **1502**A-**1502**N and graphics processor **1508** can use embedded memory modules **1518** as a shared Last Level Cache.
(246) The processor cores **1502**A-**1502**N may, for example, be homogenous cores executing the same instruction set architecture. Alternatively, the processor cores **1502**A-**1502**N are heterogeneous in terms of instruction set architecture (ISA), where one or more of processor cores **1502**A-**1502**N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set. The processor cores **1502**A-**1502**N may be heterogeneous in terms of microarchitecture, where one or more cores having a relatively higher power consumption couple with one or more power cores having a lower power consumption. As another example, the processor cores **1502**A-**1502**N are heterogeneous in terms of computational capability. Additionally, processor **1500** can be implemented on one or more chips or as an SoC integrated circuit having the illustrated components, in addition to other components.
(247) FIG. **15**B is a block diagram of hardware logic of a graphics processor core **1519**, according to some embodiments described herein. The graphics processor core **1519**, sometimes referred to as a core slice, can be one or multiple graphics cores within a modular graphics processor. The graphics processor core **1519** is exemplary of one graphics core slice, and a graphics processor as

described herein may include multiple graphics core slices based on target power and performance envelopes. Each graphics processor core **1519** can include a fixed function block **1530** coupled with multiple sub-cores **1521**A-**1521**F, also referred to as sub-slices, that include modular blocks of general-purpose and fixed function logic.

(248) The fixed function block **1530** may include a geometry/fixed function pipeline **1531** that can be shared by all sub-cores in the graphics processor core **1519**, for example, in lower performance and/or lower power graphics processor implementations. The geometry/fixed function pipeline **1531** may include a 3D fixed function pipeline (e.g., 3D pipeline **1612** as in FIG. **16**A described below) a video front-end unit, a thread spawner and thread dispatcher, and a unified return buffer manager, which manages unified return buffers (e.g., unified return buffer **1718** in FIG. **17**, as described below).

(249) The fixed function block **1530** may also include a graphics SoC interface **1532**, a graphics microcontroller **1533**, and a media pipeline **1534**. The graphics SoC interface **1532** provides an interface between the graphics processor core **1519** and other processor cores within a system on a chip integrated circuit. The graphics microcontroller **1533** is a programmable sub-processor that is configurable to manage various functions of the graphics processor core **1519**, including thread dispatch, scheduling, and pre-emption. The media pipeline **1534** (e.g., media pipeline **1616** of FIG. **16**A and FIG. **17**) includes logic to facilitate the decoding, encoding, pre-processing, and/or post-processing of multimedia data, including image and video data. The media pipeline **1534** implement media operations via requests to compute or sampling logic within the sub-cores **1521**-**1521**F.

(250) The SoC interface **1532** may enable the graphics processor core **1519** to communicate with general-purpose application processor cores (e.g., CPUs) and/or other components within an SoC, including memory hierarchy elements such as a shared last level cache memory, the system RAM, and/or embedded on-chip or on-package DRAM. The SoC interface **1532** can also enable communication with fixed function devices within the SoC, such as camera imaging pipelines, and enables the use of and/or implements global memory atomics that may be shared between the graphics processor core **1519** and CPUs within the SoC. The SoC interface **1532** can also implement power management controls for the graphics processor core **1519** and enable an interface between a clock domain of the graphics processor core **1519** and other clock domains within the SoC. Optionally, the SoC interface **1532** enables receipt of command buffers from a command streamer and global thread dispatcher that are configured to provide commands and instructions to each of one or more graphics cores within a graphics processor. The commands and instructions can be dispatched to the media pipeline **1534**, when media operations are to be performed, or a geometry and fixed function pipeline (e.g., geometry and fixed function pipeline **1531**, geometry and fixed function pipeline **1537**) when graphics processing operations are to be performed.

(251) The graphics microcontroller **1533** can be configured to perform various scheduling and management tasks for the graphics processor core **1519**. In one configuration the graphics microcontroller **1533** can, for example, perform graphics and/or compute workload scheduling on the various graphics parallel engines within execution unit (EU) arrays **1522**A-**1522**F, **1524**A-**1524**F within the sub-cores **1521**A-**1521**F. In this workload scheduling, host software executing on a CPU core of an SoC including the graphics processor core **1519** can submit workloads to one of multiple graphic processor doorbells, which invokes a scheduling operation on the appropriate graphics engine. Scheduling operations include determining which workload to run next, submitting a workload to a command streamer, pre-empting existing workloads running on an engine, monitoring progress of a workload, and notifying host software when a workload is complete. Optionally, the graphics microcontroller **1533** can also facilitate low-power or idle states for the graphics processor core **1519**, providing the graphics processor core **1519** with the ability to save and restore registers within the graphics processor core **1519** across low-power state

transitions independently from the operating system and/or graphics driver software on the system.

(252) The graphics processor core **1519** may have more than or fewer than the illustrated sub-cores **1521**A-**1521**F, up to N modular sub-cores. For each set of N sub-cores, the graphics processor core **1519** can also include shared function logic **1535**, shared and/or cache memory **1536**, a geometry/fixed function pipeline **1537**, as well as additional fixed function logic **1538** to accelerate various graphics and compute processing operations. The shared function logic **1535** can include logic units associated with the shared function logic **1720** of FIG. **17** (e.g., sampler, math, and/or inter-thread communication logic) that can be shared by each N sub-cores within the graphics processor core **1519**. The shared and/or cache memory **1536** can be a last-level cache for the set of N sub-cores **1521**A-**1521**F within the graphics processor core **1519**, and can also serve as shared memory that is accessible by multiple sub-cores. The geometry/fixed function pipeline **1537** can be included instead of the geometry/fixed function pipeline **1531** within the fixed function block **1530** and can include the same or similar logic units.

(253) The graphics processor core **1519** may include additional fixed function logic **1538** that can include various fixed function acceleration logic for use by the graphics processor core **1519**. Optionally, the additional fixed function logic **1538** includes an additional geometry pipeline for use in position only shading. In position-only shading, two geometry pipelines exist, the full geometry pipeline within the geometry/fixed function pipeline **1538**, **1531**, and a cull pipeline, which is an additional geometry pipeline which may be included within the additional fixed function logic **1538**. For example, the cull pipeline may be a trimmed down version of the full geometry pipeline. The full pipeline and the cull pipeline can execute different instances of the same application, each instance having a separate context. Position only shading can hide long cull runs of discarded triangles, enabling shading to be completed earlier in some instances. For example, the cull pipeline logic within the additional fixed function logic **1538** can execute position shaders in parallel with the main application and generally generates critical results faster than the full pipeline, as the cull pipeline fetches and shades only the position attribute of the vertices, without performing rasterization and rendering of the pixels to the frame buffer. The cull pipeline can use the generated critical results to compute visibility information for all the triangles without regard to whether those triangles are culled. The full pipeline (which in this instance may be referred to as a replay pipeline) can consume the visibility information to skip the culled triangles to shade only the visible triangles that are finally passed to the rasterization phase.

(254) Optionally, the additional fixed function logic **1538** can also include machine-learning acceleration logic, such as fixed function matrix multiplication logic, for implementations including optimizations for machine learning training or inferencing.

(255) Within each graphics sub-core **1521**A-**1521**F a set of execution resources is included that may be used to perform graphics, media, and compute operations in response to requests by graphics pipeline, media pipeline, or shader programs. The graphics sub-cores **1521**A-**1521**F include multiple EU arrays **1522**A-**1522**F, **1524**A-**1524**F, thread dispatch and inter-thread communication (TD/IC) logic **1523**A-**1523**F, a 3D (e.g., texture) sampler **1525**A-**1525**F, a media sampler **1506**A-**1506**F, a shader processor **1527**A-**1527**F, and shared local memory (SLM) **1528**A-**1528**F. The EU arrays **1522**A-**1522**F, **1524**A-**1524**F each include multiple execution units, which are general-purpose graphics processing units capable of performing floating-point and integer/fixed-point logic operations in service of a graphics, media, or compute operation, including graphics, media, or compute shader programs. The TD/IC logic **1523**A-**1523**F performs local thread dispatch and thread control operations for the execution units within a sub-core and facilitate communication between threads executing on the execution units of the sub-core. The 3D sampler **1525**A-**1525**F can read texture or other 3D graphics related data into memory. The 3D sampler can read texture data differently based on a configured sample state and the texture format associated with a given texture. The media sampler **1506**A-**1506**F can perform similar read operations based on the type and format associated with media data. For example, each graphics

sub-core **1521**A-**1521**F can alternately include a unified 3D and media sampler. Threads executing on the execution units within each of the sub-cores **1521**A-**1521**F can make use of shared local memory **1528**A-**1528**F within each sub-core, to enable threads executing within a thread group to execute using a common pool of on-chip memory.

(256) FIG. **15**C is a block diagram of general-purpose graphics processing unit (GPGPU) **1570** that can be configured as a graphics processor, e.g. the graphics processor **1508**, and/or compute accelerator, according to embodiments described herein. The GPGPU **1570** can interconnect with host processors (e.g., one or more CPU(s) **1546**) and memory **1571**, **1572** via one or more system and/or memory busses. Memory **1571** may be system memory that can be shared with the one or more CPU(s) **1546**, while memory **1572** is device memory that is dedicated to the GPGPU **1570**. For example, components within the GPGPU **1570** and device memory **1572** may be mapped into memory addresses that are accessible to the one or more CPU(s) **1546**. Access to memory **1571** and **1572** may be facilitated via a memory controller **1568**. The memory controller **1568** may include an internal direct memory access (DMA) controller **1569** or can include logic to perform operations that would otherwise be performed by a DMA controller.

(257) The GPGPU **1570** includes multiple cache memories, including an L2 cache **1553**, L1 cache **1554**, an instruction cache **1555**, and shared memory **1556**, at least a portion of which may also be partitioned as a cache memory. The GPGPU **1570** also includes multiple compute units **1560**A-**1560**N. Each compute unit **1560**A-**1560**N includes a set of vector registers **1561**, scalar registers **1562**, vector logic units **1563**, and scalar logic units **1564**. The compute units **1560**A-**1560**N can also include local shared memory **1565** and a program counter **1566**. The compute units **1560**A-**1560**N can couple with a constant cache **1567**, which can be used to store constant data, which is data that will not change during the run of kernel or shader program that executes on the GPGPU **1570**. The constant cache **1567** may be a scalar data cache and cached data can be fetched directly into the scalar registers **1562**.

(258) During operation, the one or more CPU(s) **1546** can write commands into registers or memory in the GPGPU **1570** that has been mapped into an accessible address space. The command processors **1557** can read the commands from registers or memory and determine how those commands will be processed within the GPGPU **1570**. A thread dispatcher **1558** can then be used to dispatch threads to the compute units **1560**A-**1560**N to perform those commands. Each compute unit **1560**A-**1560**N can execute threads independently of the other compute units. Additionally, each compute unit **1560**A-**1560**N can be independently configured for conditional computation and can conditionally output the results of computation to memory. The command processors **1557** can interrupt the one or more CPU(s) **1546** when the submitted commands are complete.

(259) FIG. **16**A-**16**C illustrate block diagrams of additional graphics processor and compute accelerator architectures provided by embodiments described herein, e.g. in accordance with FIG. **15**A-**15**C. The elements of FIG. **16**A-**16**C having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

(260) FIG. **16**A is a block diagram of a graphics processor **1600**, which may be a discrete graphics processing unit, or may be a graphics processor integrated with a plurality of processing cores, or other semiconductor devices such as, but not limited to, memory devices or network interfaces. The graphics processor **1600** may be a variant of the graphics processor **1508** and may be used in place of the graphics processor **1508**. Therefore, the disclosure of any features in combination with the graphics processor **1508** herein also discloses a corresponding combination with the graphics processor **1600**, but is not limited to such. The graphics processor may communicate via a memory mapped I/O interface to registers on the graphics processor and with commands placed into the processor memory. Graphics processor **1600** may include a memory interface **1614** to access memory. Memory interface **1614** can be an interface to local memory, one or more internal caches,

one or more shared external caches, and/or to system memory.
(261) Optionally, graphics processor **1600** also includes a display controller **1602** to drive display output data to a display device **1618**. Display controller **1602** includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. The display device **1618** can be an internal or external display device. In one embodiment the display device **1618** is a head mounted display device, such as a virtual reality (VR) display device or an augmented reality (AR) display device. Graphics processor **1600** may include a video codec engine **1606** to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, H.265/HEVC, Alliance for Open Media (AOMedia) VP8, VP9, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.
(262) Graphics processor **1600** may include a block image transfer (BLIT) engine **1604** to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, alternatively, 2D graphics operations may be performed using one or more components of graphics processing engine (GPE) **1610**. In some embodiments, GPE **1610** is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.
(263) GPE **1610** may include a 3D pipeline **1612** for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline **1612** includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system **1615**. While 3D pipeline **1612** can be used to perform media operations, an embodiment of GPE **1610** also includes a media pipeline **1616** that is specifically used to perform media operations, such as video post-processing and image enhancement.
(264) Media pipeline **1616** may include fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of video codec engine **1606**. Media pipeline **1616** may additionally include a thread spawning unit to spawn threads for execution on 3D/Media sub-system **1615**. The spawned threads perform computations for the media operations on one or more graphics execution units included in 3D/Media sub-system **1615**.
(265) The 3D/Media subsystem **1615** may include logic for executing threads spawned by 3D pipeline **1612** and media pipeline **1616**. The pipelines may send thread execution requests to 3D/Media subsystem **1615**, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. The 3D/Media subsystem **1615** may include one or more internal caches for thread instructions and data. Additionally, the 3D/Media subsystem **1615** may also include shared memory, including registers and addressable memory, to share data between threads and to store output data.
(266) FIG. **16**B illustrates a graphics processor **1620**, being a variant of the graphics processor **1600** and may be used in place of the graphics processor **1600** and vice versa. Therefore, the disclosure of any features in combination with the graphics processor **1600** herein also discloses a corresponding combination with the graphics processor **1620**, but is not limited to such. The graphics processor **1620** has a tiled architecture, according to embodiments described herein. The graphics processor **1620** may include a graphics processing engine cluster **1622** having multiple instances of the graphics processing engine **1610** of FIG. **16**A within a graphics engine tile **1610**A-**1610**D. Each graphics engine tile **1610**A-**1610**D can be interconnected via a set of tile interconnects **1623**A-**1623**F. Each graphics engine tile **1610**A-**1610**D can also be connected to a memory module or memory device **1626**A-**1626**D via memory interconnects **1625**A-**1625**D. The

memory devices **1626**A-**1626**D can use any graphics memory technology. For example, the memory devices **1626**A-**1626**D may be graphics double data rate (GDDR) memory. The memory devices **1626**A-**1626**D may be high-bandwidth memory (HBM) modules that can be on-die with their respective graphics engine tile **1610**A-**1610**D. The memory devices **1626**A-**1626**D may be stacked memory devices that can be stacked on top of their respective graphics engine tile **1610**A-**1610**D. Each graphics engine tile **1610**A-**1610**D and associated memory **1626**A-**1626**D may reside on separate chiplets, which are bonded to a base die or base substrate, as described in further detail in FIG. **24**B-**24**D.

(267) The graphics processor **1620** may be configured with a non-uniform memory access (NUMA) systemin which memory devices **1626**A-**1626**D are coupled with associated graphics engine tiles **1610**A-**1610**D. A given memory device may be accessed by graphics engine tiles other than the tile to which it is directly connected. However, access latency to the memory devices **1626**A-**1626**D may be lowest when accessing a local tile. In one embodiment, a cache coherent NUMA (ccNUMA) system is enabled that uses the tile interconnects **1623**A-**1623**F to enable communication between cache controllers within the graphics engine tiles **1610**A-**1610**D to keep a consistent memory image when more than one cache stores the same memory location.

(268) The graphics processing engine cluster **1622** can connect with an on-chip or on-package fabric interconnect **1624**. The fabric interconnect **1624** can enable communication between graphics engine tiles **1610**A-**1610**D and components such as the video codec **1606** and one or more copy engines **1604**. The copy engines **1604** can be used to move data out of, into, and between the memory devices **1626**A-**1626**D and memory that is external to the graphics processor **1620** (e.g., system memory). The fabric interconnect **1624** can also be used to interconnect the graphics engine tiles **1610**A-**1610**D. The graphics processor **1620** may optionally include a display controller **1602** to enable a connection with an external display device **1618**. The graphics processor may also be configured as a graphics or compute accelerator. In the accelerator configuration, the display controller **1602** and display device **1618** may be omitted.

(269) The graphics processor **1620** can connect to a host system via a host interface **1628**. The host interface **1628** can enable communication between the graphics processor **1620**, system memory, and/or other system components. The host interface **1628** can be, for example, a PCI express bus or another type of host system interface.

(270) FIG. **16**C illustrates a compute accelerator **1630**, according to embodiments described herein. The compute accelerator **1630** can include architectural similarities with the graphics processor **1620** of FIG. **16**B and is optimized for compute acceleration. A compute engine cluster **1632** can include a set of compute engine tiles **1640**A-**1640**D that include execution logic that is optimized for parallel or vector-based general-purpose compute operations. The compute engine tiles **1640**A-**1640**D may not include fixed function graphics processing logic, although in some embodiments one or more of the compute engine tiles **1640**A-**1640**D can include logic to perform media acceleration. The compute engine tiles **1640**A-**1640**D can connect to memory **1626**A-**1626**D via memory interconnects **1625**A-**1625**D. The memory **1626**A-**1626**D and memory interconnects **1625**A-**1625**D may be similar technology as in graphics processor **1620**, or can be different. The graphics compute engine tiles **1640**A-**1640**D can also be interconnected via a set of tile interconnects **1623**A-**1623**F and may be connected with and/or interconnected by a fabric interconnect **1624**. In one embodiment the compute accelerator **1630** includes a large L3 cache **1636** that can be configured as a device-wide cache. The compute accelerator **1630** can also connect to a host processor and memory via a host interface **1628** in a similar manner as the graphics processor **1620** of FIG. **16**B.

(271) Graphics Processing Engine

(272) FIG. **17** is a block diagram of a graphics processing engine **1710** of a graphics processor in accordance with some embodiments. The graphics processing engine (GPE) **1710** may be a version of the GPE **1610** shown in FIG. **16**A, and may also represent a graphics engine tile **1610**A-**1610**D

of FIG. **16**B. The elements of FIG. **17** having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. For example, the 3D pipeline **1612** and media pipeline **1616** of FIG. **16**A are also illustrated in FIG. **17**. The media pipeline **1616** is optional in some embodiments of the GPE **1710** and may not be explicitly included within the GPE **1710**. For example and in at least one embodiment, a separate media and/or image processor is coupled to the GPE **1710**.

(273) GPE **1710** may couple with or include a command streamer **1703**, which provides a command stream to the 3D pipeline **1612** and/or media pipelines **1616**. Alternatively or additionally, the command streamer **1703** may be directly coupled to a unified return buffer **1718**. The unified return buffer **1718** may be communicatively coupled to a graphics core array **1714**. Optionally, the command streamer **1703** is coupled with memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer **1703** may receive commands from the memory and sends the commands to 3D pipeline **1612** and/or media pipeline **1616**. The commands are directives fetched from a ring buffer, which stores commands for the 3D pipeline **1612** and media pipeline **1616**. The ring buffer can additionally include batch command buffers storing batches of multiple commands. The commands for the 3D pipeline **1612** can also include references to data stored in memory, such as but not limited to vertex and geometry data for the 3D pipeline **1612** and/or image data and memory objects for the media pipeline **1616**. The 3D pipeline **1612** and media pipeline **1616** process the commands and data by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the graphics core array **1714**. The graphics core array **1714** may include one or more blocks of graphics cores (e.g., graphics core(s) **1715**A, graphics core(s) **1715**B), each block including one or more graphics cores. Each graphics core includes a set of graphics execution resources that includes general-purpose and graphics specific execution logic to perform graphics and compute operations, as well as fixed function texture processing and/or machine learning and artificial intelligence acceleration logic.

(274) In various embodiments the 3D pipeline **1612** can include fixed function and programmable logic to process one or more shader programs, such as vertex shaders, geometry shaders, pixel shaders, fragment shaders, compute shaders, or other shader programs, by processing the instructions and dispatching execution threads to the graphics core array **1714**. The graphics core array **1714** provides a unified block of execution resources for use in processing these shader programs. Multi-purpose execution logic (e.g., execution units) within the graphics core(s) **1715**A-**1714**B of the graphics core array **1714** includes support for various 3D API shader languages and can execute multiple simultaneous execution threads associated with multiple shaders.

(275) The graphics core array **1714** may include execution logic to perform media functions, such as video and/or image processing. The execution units may include general-purpose logic that is programmable to perform parallel general-purpose computational operations, in addition to graphics processing operations. The general-purpose logic can perform processing operations in parallel or in conjunction with general-purpose logic within the processor core(s) **1407** of FIG. **14** or core **1502**A-**1502**N as in FIG. **15**A.

(276) Output data generated by threads executing on the graphics core array **1714** can output data to memory in a unified return buffer (URB) **1718**. The URB **1718** can store data for multiple threads. The URB **1718** may be used to send data between different threads executing on the graphics core array **1714**. The URB **1718** may additionally be used for synchronization between threads on the graphics core array **1714** and fixed function logic within the shared function logic **1720**.

(277) Optionally, the graphics core array **1714** may be scalable, such that the array includes a variable number of graphics cores, each having a variable number of execution units based on the

target power and performance level of GPE **1710**. The execution resources may be dynamically scalable, such that execution resources may be enabled or disabled as needed.
(278) The graphics core array **1714** couples with shared function logic **1720** that includes multiple resources that are shared between the graphics cores in the graphics core array. The shared functions within the shared function logic **1720** are hardware logic units that provide specialized supplemental functionality to the graphics core array **1714**. In various embodiments, shared function logic **1720** includes but is not limited to sampler **1721**, math **1722**, and inter-thread communication (ITC) **1723** logic. Additionally, one or more cache(s) **1725** within the shared function logic **1720** may be implemented.
(279) A shared function is implemented at least in a case where the demand for a given specialized function is insufficient for inclusion within the graphics core array **1714**. Instead a single instantiation of that specialized function is implemented as a stand-alone entity in the shared function logic **1720** and shared among the execution resources within the graphics core array **1714**. The precise set of functions that are shared between the graphics core array **1714** and included within the graphics core array **1714** varies across embodiments. Specific shared functions within the shared function logic **1720** that are used extensively by the graphics core array **1714** may be included within shared function logic **1716** within the graphics core array **1714**. Optionally, the shared function logic **1716** within the graphics core array **1714** can include some or all logic within the shared function logic **1720**. All logic elements within the shared function logic **1720** may be duplicated within the shared function logic **1716** of the graphics core array **1714**. Alternatively, the shared function logic **1720** is excluded in favor of the shared function logic **1716** within the graphics core array **1714**.
(280) Execution Units
(281) FIG. **18**A-**18**B illustrate thread execution logic **1800** including an array of processing elements employed in a graphics processor core according to embodiments described herein. The elements of FIG. **18**A-**18**B having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. FIG. **18**A-**18**B illustrates an overview of thread execution logic **1800**, which may be representative of hardware logic illustrated with each sub-core **1521**A-**1521**F of FIG. **15**B. FIG. **18**A is representative of an execution unit within a general-purpose graphics processor, while FIG. **18**B is representative of an execution unit that may be used within a compute accelerator.
(282) As illustrated in FIG. **18**A, thread execution logic **1800** may include a shader processor **1802**, a thread dispatcher **1804**, instruction cache **1806**, a scalable execution unit array including a plurality of execution units **1808**A-**1808**N, a sampler **1810**, shared local memory **1811**, a data cache **1812**, and a data port **1814**. Optionally, the scalable execution unit array can dynamically scale by enabling or disabling one or more execution units (e.g., any of execution units **1808**A, **1808**B, **1808**C, **1808**D, through **1808**N-**1** and **1808**N) based on the computational requirements of a workload. The included components may be interconnected via an interconnect fabric that links to each of the components. Thread execution logic **1800** may include one or more connections to memory, such as system memory or cache memory, through one or more of instruction cache **1806**, data port **1814**, sampler **1810**, and execution units **1808**A-**1808**N. Each execution unit (e.g. **1808**A) may be a stand-alone programmable general-purpose computational unit that is capable of executing multiple simultaneous hardware threads while processing multiple data elements in parallel for each thread. In various embodiments, the array of execution units **1808**A-**1808**N is scalable to include any number individual execution units.
(283) The execution units **1808**A-**1808**N may be primarily used to execute shader programs. A shader processor **1802** can process the various shader programs and dispatch execution threads associated with the shader programs via a thread dispatcher **1804**. The thread dispatcher may

include logic to arbitrate thread initiation requests from the graphics and media pipelines and instantiate the requested threads on one or more execution units **1808**A-**1808**N. For example, a geometry pipeline can dispatch vertex, tessellation, or geometry shaders to the thread execution logic for processing. Optionally, the thread dispatcher **1804** can also process runtime thread spawning requests from the executing shader programs.

(284) The execution units **1808**A-**1808**N may support an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders). Each of the execution units **1808**A-**1808**N is capable of multi-issue single instruction multiple data (SIMD) execution and multi-threaded operation enables an efficient execution environment in the face of higher latency memory accesses. Each hardware thread within each execution unit has a dedicated high-bandwidth register file and associated independent thread-state. Execution is multi-issue per clock to pipelines capable of integer, single and double precision floating point operations, SIMD branch capability, logical operations, transcendental operations, and other miscellaneous operations. While waiting for data from memory or one of the shared functions, dependency logic within the execution units **1808**A-**1808**N causes a waiting thread to sleep until the requested data has been returned. While the waiting thread is sleeping, hardware resources may be devoted to processing other threads. For example, during a delay associated with a vertex shader operation, an execution unit can perform operations for a pixel shader, fragment shader, or another type of shader program, including a different vertex shader, such as vertex shader **2107** illustrated in FIG. **21**. Various embodiments can apply to use execution by use of Single Instruction Multiple Thread (SIMT) as an alternate to use of SIMD or in addition to use of SIMD. Reference to a SIMD core or operation can apply also to SIMT or apply to SIMD in combination with SIMT.

(285) Each execution unit in execution units **1808**A-**1808**N operates on arrays of data elements. The number of data elements is the “execution size,” or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical Arithmetic Logic Units (ALUs), Floating-Point Units (FPUs), or other logic units (e.g., tensor cores, ray tracing cores, etc.) for a particular graphics processor. Additionally, the execution units **1808**A-**1808**N may support integer and floating-point data types.

(286) The execution unit instruction set includes SIMD instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (Quad-Word (QW) size data elements), eight separate 32-bit packed data elements (Double Word (DW) size data elements), sixteen separate 16-bit packed data elements (Word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

(287) Optionally, one or more execution units can be combined into a fused execution unit **1809**A-**1809**N having thread control logic (**1807**A-**1807**N) that is common to the fused EUs. Multiple EUs can be fused into an EU group. Each EU in the fused EU group can be configured to execute a separate SIMD hardware thread. The number of EUs in a fused EU group can vary according to embodiments. Additionally, various SIMD widths can be performed per-EU, including but not limited to SIMD8, SIMD16, and SIMD32. Each fused graphics execution unit **1809**A-**1809**N includes at least two execution units. For example, fused execution unit **1809**A includes a first EU **1808**A, second EU **1808**B, and thread control logic **1807**A that is common to the first EU **1808**A and the second EU **1808**B. The thread control logic **1807**A controls threads executed on the fused

graphics execution unit **1809**A, allowing each EU within the fused execution units **1809**A-**1809**N to execute using a common instruction pointer register.

(288) One or more internal instruction caches (e.g., **1806**) are included in the thread execution logic **1800** to cache thread instructions for the execution units. One or more data caches (e.g., **1812**) may be included in the thread execution logic **1800** to cache thread data during thread execution. Threads executing on the execution logic **1800** can also store explicitly managed data in the shared local memory **1811**. A sampler **1810** may be included to provide texture sampling for 3D operations and media sampling for media operations. Sampler **1810** may include specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

(289) During execution, the graphics and media pipelines send thread initiation requests to thread execution logic **1800** via thread spawning and dispatch logic. Once a group of geometric objects has been processed and rasterized into pixel data, pixel processor logic (e.g., pixel shader logic, fragment shader logic, etc.) within the shader processor **1802** is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). A pixel shader or fragment shader may calculate the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel processor logic within the shader processor **1802** may then execute an application programming interface (API)-supplied pixel or fragment shader program. To execute the shader program, the shader processor **1802** dispatches threads to an execution unit (e.g., **1808**A) via thread dispatcher **1804**. Shader processor **1802** may use texture sampling logic in the sampler **1810** to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

(290) In addition, the data port **1814** may provide a memory access mechanism for the thread execution logic **1800** to output processed data to memory for further processing on a graphics processor output pipeline. The data port **1814** may include or couple to one or more cache memories (e.g., data cache **1812**) to cache data for memory access via the data port **1814**.

(291) Optionally, the execution logic **1800** can also include a ray tracer **1805** that can provide ray tracing acceleration functionality. The ray tracer **1805** can support a ray tracing instruction set that includes instructions/functions for ray generation. The ray tracing instruction set can be similar to or different from the ray-tracing instruction set supported by the ray tracing cores **372** in FIG. **3**C.

(292) FIG. **18**B illustrates exemplary internal details of an execution unit **1808**. A graphics execution unit **1808** can include an instruction fetch unit **1837**, a general register file array (GRF) **1824**, an architectural register file array (ARF) **1826**, a thread arbiter **1822**, a send unit **1830**, a branch unit **1832**, a set of SIMD floating point units (FPUs) **1834**, and optionally a set of dedicated integer SIMD ALUs **1835**. The GRF **1824** and ARF **1826** includes the set of general register files and architecture register files associated with each simultaneous hardware thread that may be active in the graphics execution unit **1808**. Per thread architectural state may be maintained in the ARF **1826**, while data used during thread execution is stored in the GRF **1824**. The execution state of each thread, including the instruction pointers for each thread, can be held in thread-specific registers in the ARF **1826**.

(293) The graphics execution unit **1808** may have an architecture that is a combination of Simultaneous Multi-Threading (SMT) and fine-grained Interleaved Multi-Threading (IMT). The architecture may have a modular configuration that can be fine-tuned at design time based on a target number of simultaneous threads and number of registers per execution unit, where execution unit resources are divided across logic used to execute multiple simultaneous threads. The number of logical threads that may be executed by the graphics execution unit **1808** is not limited to the number of hardware threads, and multiple logical threads can be assigned to each hardware thread.

(294) Optionally, the graphics execution unit **1808** can co-issue multiple instructions, which may

each be different instructions. The thread arbiter **1822** of the graphics execution unit **1808** can dispatch the instructions to one of the send unit **1830**, branch unit **1832**, or SIMD FPU(s) **1834** for execution. Each execution thread can access **128** general-purpose registers within the GRF **1824**, where each register can store 32 bytes, accessible as a SIMD 8-element vector of 32-bit data elements. Each execution unit thread may have access to 4 Kbytes within the GRF **1824**, although embodiments are not so limited, and greater or fewer register resources may be provided in other embodiments. The graphics execution unit **1808** may be partitioned into seven hardware threads that can independently perform computational operations, although the number of threads per execution unit can also vary according to embodiments, for example, up to 16 hardware threads may be supported. In an exemplary embodiment, in which seven threads may access 4 Kbytes, the GRF **1824** can store a total of 28 Kbytes. In another exemplary embodiment, where 16 threads may access 4 Kbytes, the GRF **1824** can store a total of 64 Kbytes. The number of threads per execution unit are, however, not limited to those examples and may be more or less than the given numbers. Flexible addressing modes can permit registers to be addressed together to build effectively wider registers or to represent strided rectangular block data structures.
(295) Additionally or alternatively, memory operations, sampler operations, and other longer-latency system communications may be dispatched via "send" instructions that are executed by the message passing send unit **1830**. Branch instructions may be dispatched to a dedicated branch unit **1832** to facilitate SIMD divergence and eventual convergence.
(296) The graphics execution unit **1808** may include one or more SIMD floating point units (FPU(s)) **1834** to perform floating-point operations. The FPU(s) **1834** may also support integer computation. In some instances, the FPU(s) **1834** can SIMD execute up to M number of 32-bit floating-point (or integer) operations, or SIMD execute up to 2M 16-bit integer or 16-bit floating-point operations. Optionally, at least one of the FPU(s) provides extended math capability to support high-throughput transcendental math functions and double precision 64-bit floating-point. A set of 8-bit integer SIMD ALUs **1835** may also be present, and may be specifically optimized to perform operations associated with machine learning computations.
(297) Optionally, arrays of multiple instances of the graphics execution unit **1808** can be instantiated in a graphics sub-core grouping (e.g., a sub-slice). For scalability, product architects can choose the exact number of execution units per sub-core grouping. The graphics execution unit **1808** may execute instructions across a plurality of execution channels. In addition, each thread executed on the graphics execution unit **1808** may be executed on a different channel.
(298) FIG. **19** illustrates a further exemplary execution unit **1900**. The elements of FIG. **19** having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. The execution unit **1900** may be a compute-optimized execution unit for use in, for example, a compute engine tile **1640**A-**1640**D as in FIG. **16**C, but is not limited as such. The execution unit **1900** may also be used in a graphics engine tile **1610**A-**1610**D as in FIG. **16**B. The execution unit **1900** may include a thread control unit **1901**, a thread state unit **1902**, an instruction fetch/prefetch unit **1903**, and an instruction decode unit **1904**. The execution unit **1900** may additionally include a register file **1906** that stores registers that can be assigned to hardware threads within the execution unit. The execution unit **1900** may additionally include a send unit **1907** and a branch unit **1908**. The send unit **1907** and branch unit **1908** may operate similarly as the send unit **1830** and a branch unit **1832** of the graphics execution unit **1808** of FIG. **18**B.
(299) The execution unit **1900** can also include a compute unit **1910** that includes multiple different types of functional units. The compute unit **1910** may also include an ALU unit **1911** that includes an array of arithmetic logic units. The ALU unit **1911** can be configured to perform 64-bit, 32-bit, and 16-bit integer and floating-point operations. Integer and floating-point operations may be performed simultaneously. The compute unit **1910** can also include a systolic array **1912**, and a

math unit **1913**. The systolic array **1912** includes a W wide and D deep network of data processing units that can be used to perform vector or other data-parallel operations in a systolic manner. The systolic array **1912** can be configured to perform matrix operations, such as matrix dot product operations. The systolic array **1912** may support 16-bit floating point operations, as well as 8-bit and 4-bit integer operations. The systolic array **1912** may be configured to accelerate machine learning operations. The systolic array **1912** can be configured with support for the bfloat16, a 16-bit floating point format. A math unit **1913** can be included to perform a specific subset of mathematical operations in an efficient and lower-power manner than then ALU unit **1911**. The math unit **1913** can include math logic found in shared function logic of a graphics processing engine provided by other embodiments described, e.g., the math logic **1722** of the shared function logic **1720** of FIG. **17**. The math unit **1913** can be configured to perform 32-bit and 64-bit floating point operations.

(300) The thread control unit **1901** includes logic to control the execution of threads within the execution unit. The thread control unit **1901** can include thread arbitration logic to start, stop, and preempt execution of threads within the execution unit **1900**. The thread state unit **1902** can be used to store thread state for threads assigned to execute on the execution unit **1900**. Storing the thread state within the execution unit **1900** enables the rapid pre-emption of threads when those threads become blocked or idle. The instruction fetch/prefetch unit **1903** can fetch instructions from an instruction cache of higher-level execution logic (e.g., instruction cache **1806** as in FIG. **18**A). The instruction fetch/prefetch unit **1903** can also issue prefetch requests for instructions to be loaded into the instruction cache based on an analysis of currently executing threads. The instruction decode unit **1904** can be used to decode instructions to be executed by the compute units. The instruction decode unit **1904** can be used as a secondary decoder to decode complex instructions into constituent micro-operations.

(301) The execution unit **1900** additionally includes a register file **1906** that can be used by hardware threads executing on the execution unit **1900**. Registers in the register file **1906** can be divided across the logic used to execute multiple simultaneous threads within the compute unit **1910** of the execution unit **1900**. The number of logical threads that may be executed by the graphics execution unit **1900** is not limited to the number of hardware threads, and multiple logical threads can be assigned to each hardware thread. The size of the register file **1906** can vary across embodiments based on the number of supported hardware threads. Register renaming may be used to dynamically allocate registers to hardware threads.

(302) FIG. **20** is a block diagram illustrating graphics processor instruction format **2000**. The graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. The graphics processor instruction formats **2000** described and illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

(303) The graphics processor execution units as described herein may natively support instructions in a 128-bit instruction format **2010**. A 64-bit compacted instruction format **2030** is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit instruction format **2010** provides access to all instruction options, while some options and operations are restricted in the 64-bit format **2030**. The native instructions available in the 64-bit format **2030** vary by embodiment. The instruction is compacted in part using a set of index values in an index field **2013**. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit instruction format **2010**. Other sizes and formats of instruction can be used.

(304) For each format, instruction opcode **2012** defines the operation that the execution unit is to

perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. Instruction control field **2014** may enable control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For instructions in the 128-bit instruction format **2010** an exec-size field **2016** limits the number of data channels that will be executed in parallel. An exec-size field **2016** may not be available for use in the 64-bit compact instruction format **2030**.

(305) Some execution unit instructions have up to three operands including two source operands, src0 **2020**, src1 **2022**, and one destination **2018**. The execution units may support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 **2024**), where the instruction opcode **2012** determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

(306) The 128-bit instruction format **2010** may include an access/address mode field **2026** specifying, for example, whether direct register addressing mode or indirect register addressing mode is used. When direct register addressing mode is used, the register address of one or more operands is directly provided by bits in the instruction.

(307) The 128-bit instruction format **2010** may also include an access/address mode field **2026**, which specifies an address mode and/or an access mode for the instruction. The access mode may be used to define a data access alignment for the instruction. Access modes including a 16-byte aligned access mode and a 1-byte aligned access mode may be supported, where the byte alignment of the access mode determines the access alignment of the instruction operands. For example, when in a first mode, the instruction may use byte-aligned addressing for source and destination operands and when in a second mode, the instruction may use 16-byte-aligned addressing for all source and destination operands.

(308) The address mode portion of the access/address mode field **2026** may determine whether the instruction is to use direct or indirect addressing. When direct register addressing mode is used bits in the instruction directly provide the register address of one or more operands. When indirect register addressing mode is used, the register address of one or more operands may be computed based on an address register value and an address immediate field in the instruction.

(309) Instructions may be grouped based on opcode **2012** bit-fields to simplify Opcode decode **2040**. For an 8-bit opcode, bits 4, 5, and 6 allow the execution unit to determine the type of opcode. The precise opcode grouping shown is merely an example. A move and logic opcode group **2042** may include data movement and logic instructions (e.g., move (mov), compare (cmp)). Move and logic group **2042** may share the five most significant bits (MSB), where move (mov) instructions are in the form of 0000xxxxb and logic instructions are in the form of 0001xxxxb. A flow control instruction group **2044** (e.g., call, jump (jmp)) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group **2046** includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group **2048** includes component-wise arithmetic instructions (e.g., add, multiply (mul)) in the form of 0100xxxxb (e.g., 0x40). The parallel math group **2048** performs the arithmetic operations in parallel across data channels. The vector math group **2050** includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands. The illustrated opcode decode **2040**, in one embodiment, can be used to determine which portion of an execution unit will be used to execute a decoded instruction. For example, some instructions may be designated as systolic instructions that will be performed by a systolic array. Other instructions, such as ray-tracing instructions (not shown) can be routed to a ray-tracing core or ray-tracing logic within a

slice or partition of execution logic.
(310) Graphics Pipeline
(311) FIG. **21** is a block diagram of graphics processor **2100**, according to another embodiment. The elements of FIG. **21** having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.
(312) The graphics processor **2100** may include different types of graphics processing pipelines, such as a geometry pipeline **2120**, a media pipeline **2130**, a display engine **2140**, thread execution logic **2150**, and a render output pipeline **2170**. Graphics processor **2100** may be a graphics processor within a multi-core processing system that includes one or more general-purpose processing cores. The graphics processor may be controlled by register writes to one or more control registers (not shown) or via commands issued to graphics processor **2100** via a ring interconnect **2102**. Ring interconnect **2102** may couple graphics processor **2100** to other processing components, such as other graphics processors or general-purpose processors. Commands from ring interconnect **2102** are interpreted by a command streamer **2103**, which supplies instructions to individual components of the geometry pipeline **2120** or the media pipeline **2130**.
(313) Command streamer **2103** may direct the operation of a vertex fetcher **2105** that reads vertex data from memory and executes vertex-processing commands provided by command streamer **2103**. The vertex fetcher **2105** may provide vertex data to a vertex shader **2107**, which performs coordinate space transformation and lighting operations to each vertex. Vertex fetcher **2105** and vertex shader **2107** may execute vertex-processing instructions by dispatching execution threads to execution units **2152**A-**2152**B via a thread dispatcher **2131**.
(314) The execution units **2152**A-**2152**B may be an array of vector processors having an instruction set for performing graphics and media operations. The execution units **2152**A-**2152**B may have an attached L1 cache **2151** that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.
(315) A geometry pipeline **2120** may include tessellation components to perform hardware-accelerated tessellation of 3D objects. A programmable hull shader **2111** may configure the tessellation operations. A programmable domain shader **2117** may provide back-end evaluation of tessellation output. A tessellator **2113** may operate at the direction of hull shader **2111** and contain special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to geometry pipeline **2120**. In addition, if tessellation is not used, tessellation components (e.g., hull shader **2111**, tessellator **2113**, and domain shader **2117**) can be bypassed.
(316) Complete geometric objects may be processed by a geometry shader **2119** via one or more threads dispatched to execution units **2152**A-**2152**B, or can proceed directly to the clipper **2129**. The geometry shader may operate on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader **2119** receives input from the vertex shader **2107**. The geometry shader **2119** may be programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.
(317) Before rasterization, a clipper **2129** processes vertex data. The clipper **2129** may be a fixed function clipper or a programmable clipper having clipping and geometry shader functions. A rasterizer and depth test component **2173** in the render output pipeline **2170** may dispatch pixel shaders to convert the geometric objects into per pixel representations. The pixel shader logic may be included in thread execution logic **2150**. Optionally, an application can bypass the rasterizer and depth test component **2173** and access un-rasterized vertex data via a stream out unit **2123**.
(318) The graphics processor **2100** has an interconnect bus, interconnect fabric, or some other

interconnect mechanism that allows data and message passing amongst the major components of the processor. In some embodiments, execution units **2152**A-**2152**B and associated logic units (e.g., L1 cache **2151**, sampler **2154**, texture cache **2158**, etc.) interconnect via a data port **2156** to perform memory access and communicate with render output pipeline components of the processor. A sampler **2154**, caches **2151**, **2158** and execution units **2152**A-**2152**B each may have separate memory access paths. Optionally, the texture cache **2158** can also be configured as a sampler cache.

(319) The render output pipeline **2170** may contain a rasterizer and depth test component **2173** that converts vertex-based objects into an associated pixel-based representation. The rasterizer logic may include a windower/masker unit to perform fixed function triangle and line rasterization. An associated render cache **2178** and depth cache **2179** are also available in some embodiments. A pixel operations component **2177** performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine **2141**, or substituted at display time by the display controller **2143** using overlay display planes. A shared L3 cache **2175** may be available to all graphics components, allowing the sharing of data without the use of main system memory.

(320) The graphics processor media pipeline **2130** may include a media engine **2137** and a video front-end **2134**. Video front-end **2134** may receive pipeline commands from the command streamer **2103**. The media pipeline **2130** may include a separate command streamer. Video front-end **2134** may process media commands before sending the command to the media engine **2137**. Media engine **2137** may include thread spawning functionality to spawn threads for dispatch to thread execution logic **2150** via thread dispatcher **2131**.

(321) The graphics processor **2100** may include a display engine **2140**. This display engine **2140** may be external to processor **2100** and may couple with the graphics processor via the ring interconnect **2102**, or some other interconnect bus or fabric. Display engine **2140** may include a 2D engine **2141** and a display controller **2143**. Display engine **2140** may contain special purpose logic capable of operating independently of the 3D pipeline. Display controller **2143** may couple with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via a display device connector.

(322) The geometry pipeline **2120** and media pipeline **2130** maybe configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). A driver software for the graphics processor may translate API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. Support may be provided for the Open Graphics Library (OpenGL), Open Computing Language (OpenCL), and/or Vulkan graphics and compute API, all from the Khronos Group. Support may also be provided for the Direct3D library from the Microsoft Corporation. A combination of these libraries may be supported. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

(323) Graphics Pipeline Programming

(324) FIG. **22**A is a block diagram illustrating a graphics processor command format **2200** used for programming graphics processing pipelines, such as, for example, the pipelines described herein in conjunction with FIG. **16**A, **17**, **21**. FIG. **22**B is a block diagram illustrating a graphics processor command sequence **2210** according to an embodiment. The solid lined boxes in FIG. **22**A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format **2200** of FIG. **22**A includes data fields to identify a client **2202**, a command operation code (opcode) **2204**, and data **2206** for the command. A sub-opcode **2205** and a command size **2208** are also included in some commands.

(325) Client **2202** may specify the client unit of the graphics device that processes the command data. A graphics processor command parser may examine the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. The graphics processor client units may include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit may have a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode **2204** and, if present, sub-opcode **2205** to determine the operation to perform. The client unit performs the command using information in data field **2206**. For some commands an explicit command size **2208** is expected to specify the size of the command. The command parser may automatically determine the size of at least some of the commands based on the command opcode. Commands may be aligned via multiples of a double word. Other command formats can also be used.
(326) The flow diagram in FIG. **22**B illustrates an exemplary graphics processor command sequence **2210**. Software or firmware of a data processing system that features an exemplary graphics processor may use a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for purposes of example only and is not limited to these specific commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in at least partially concurrence.
(327) The graphics processor command sequence **2210** may begin with a pipeline flush command **2212** to cause any active graphics pipeline to complete the currently pending commands for the pipeline. Optionally, the 3D pipeline **2222** and the media pipeline **2224** may not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. Pipeline flush command **2212** can be used for pipeline synchronization or before placing the graphics processor into a low power state.
(328) A pipeline select command **2213** may be used when a command sequence requires the graphics processor to explicitly switch between pipelines. A pipeline select command **2213** may be required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. A pipeline flush command **2212** may be required immediately before a pipeline switch via the pipeline select command **2213**.
(329) A pipeline control command **2214** may configure a graphics pipeline for operation and may be used to program the 3D pipeline **2222** and the media pipeline **2224**. The pipeline control command **2214** may configure the pipeline state for the active pipeline. The pipeline control command **2214** may be used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.
(330) Return buffer state commands **2216** may be used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. The graphics processor may also use one or more return buffers to store output data and to perform cross thread communication. The return buffer state **2216** may include selecting the size and number of return buffers to use for a set of pipeline operations.
(331) The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination **2220**, the command sequence is tailored to the 3D pipeline **2222** beginning with the 3D pipeline state **2230** or the media pipeline **2224** beginning at the media pipeline state **2240**.
(332) The commands to configure the 3D pipeline state **2230** include 3D state setting commands

for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based on the particular 3D API in use. The 3D pipeline state **2230** commands may also be able to selectively disable or bypass certain pipeline elements if those elements will not be used.

(333) A 3D primitive **2232** command may be used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive **2232** command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive **2232** command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. The 3D primitive **2232** command may be used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, 3D pipeline **2222** dispatches shader execution threads to graphics processor execution units.

(334) The 3D pipeline **2222** may be triggered via an execute **2234** command or event. A register may write trigger command executions. An execution may be triggered via a 'go' or 'kick' command in the command sequence. Command execution may be triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

(335) The graphics processor command sequence **2210** may follow the media pipeline **2224** path when performing media operations. In general, the specific use and manner of programming for the media pipeline **2224** depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. The media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general-purpose processing cores. The media pipeline may also include elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

(336) Media pipeline **2224** may be configured in a similar manner as the 3D pipeline **2222**. A set of commands to configure the media pipeline state **2240** are dispatched or placed into a command queue before the media object commands **2242**. Commands for the media pipeline state **2240** may include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. Commands for the media pipeline state **2240** may also support the use of one or more pointers to "indirect" state elements that contain a batch of state settings.

(337) Media object commands **2242** may supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. Optionally, all media pipeline states must be valid before issuing a media object command **2242**. Once the pipeline state is configured and media object commands **2242** are queued, the media pipeline **2224** is triggered via an execute command **2244** or an equivalent execute event (e.g., register write). Output from media pipeline **2224** may then be post processed by operations provided by the 3D pipeline **2222** or the media pipeline **2224**. GPGPU operations may be configured and executed in a similar manner as media operations.

(338) Graphics Software Architecture

(339) FIG. **23** illustrates an exemplary graphics software architecture for a data processing system **2300**. Such a software architecture may include a 3D graphics application **2310**, an operating system **2320**, and at least one processor **2330**. Processor **2330** may include a graphics processor

2332 and one or more general-purpose processor core(s) **2334**. The processor **2330** may be a variant of the processor **1402** or any other of the processors described herein. The processor **2330** may be used in place of the processor **1402** or any other of the processors described herein. Therefore, the disclosure of any features in combination with the processor **1402** or any other of the processors described herein also discloses a corresponding combination with the graphics processor **2330**, but is not limited to such. Moreover, the elements of FIG. **23** having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such. The graphics application **2310** and operating system **2320** are each executed in the system memory **2350** of the data processing system.

(340) 3D graphics application **2310** may contain one or more shader programs including shader instructions **2312**. The shader language instructions may be in a high-level shader language, such as the High-Level Shader Language (HLSL) of Direct3D, the OpenGL Shader Language (GLSL), and so forth. The application may also include executable instructions **2314** in a machine language suitable for execution by the general-purpose processor core **2334**. The application may also include graphics objects **2316** defined by vertex data.

(341) The operating system **2320** may be a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. The operating system **2320** can support a graphics API **2322** such as the Direct3D API, the OpenGL API, or the Vulkan API. When the Direct3D API is in use, the operating system **2320** uses a front-end shader compiler **2324** to compile any shader instructions **2312** in HLSL into a lower-level shader language. The compilation may be a just-in-time (JIT) compilation or the application can perform shader pre-compilation. High-level shaders may be compiled into low-level shaders during the compilation of the 3D graphics application **2310**. The shader instructions **2312** may be provided in an intermediate form, such as a version of the Standard Portable Intermediate Representation (SPIR) used by the Vulkan API.

(342) User mode graphics driver **2326** may contain a back-end shader compiler **2327** to convert the shader instructions **2312** into a hardware specific representation. When the OpenGL API is in use, shader instructions **2312** in the GLSL high-level language are passed to a user mode graphics driver **2326** for compilation. The user mode graphics driver **2326** may use operating system kernel mode functions **2328** to communicate with a kernel mode graphics driver **2329**. The kernel mode graphics driver **2329** may communicate with graphics processor **2332** to dispatch commands and instructions.

(343) IP Core Implementations

(344) One or more aspects may be implemented by representative code stored on a machine-readable medium which represents and/or defines logic within an integrated circuit such as a processor. For example, the machine-readable medium may include instructions which represent various logic within the processor. When read by a machine, the instructions may cause the machine to fabricate the logic to perform the techniques described herein. Such representations, known as “IP cores,” are reusable units of logic for an integrated circuit that may be stored on a tangible, machine-readable medium as a hardware model that describes the structure of the integrated circuit. The hardware model may be supplied to various customers or manufacturing facilities, which load the hardware model on fabrication machines that manufacture the integrated circuit. The integrated circuit may be fabricated such that the circuit performs operations described in association with any of the embodiments described herein.

(345) FIG. **24**A is a block diagram illustrating an IP core development system **2400** that may be used to manufacture an integrated circuit to perform operations according to an embodiment. The IP core development system **2400** may be used to generate modular, re-usable designs that can be

incorporated into a larger design or used to construct an entire integrated circuit (e.g., an SOC integrated circuit). A design facility **2430** can generate a software simulation **2410** of an IP core design in a high-level programming language (e.g., C/C++). The software simulation **2410** can be used to design, test, and verify the behavior of the IP core using a simulation model **2412**. The simulation model **2412** may include functional, behavioral, and/or timing simulations. A register transfer level (RTL) design **2415** can then be created or synthesized from the simulation model **2412**. The RTL design **2415** is an abstraction of the behavior of the integrated circuit that models the flow of digital signals between hardware registers, including the associated logic performed using the modeled digital signals. In addition to an RTL design **2415**, lower-level designs at the logic level or transistor level may also be created, designed, or synthesized. Thus, the particular details of the initial design and simulation may vary.

(346) The RTL design **2415** or equivalent may be further synthesized by the design facility into a hardware model **2420**, which may be in a hardware description language (HDL), or some other representation of physical design data. The HDL may be further simulated or tested to verify the IP core design. The IP core design can be stored for delivery to a 3.sup.rd party fabrication facility **2465** using non-volatile memory **2440** (e.g., hard disk, flash memory, or any non-volatile storage medium). Alternatively, the IP core design may be transmitted (e.g., via the Internet) over a wired connection **2450** or wireless connection **2460**. The fabrication facility **2465** may then fabricate an integrated circuit that is based at least in part on the IP core design. The fabricated integrated circuit can be configured to perform operations in accordance with at least one embodiment described herein.

(347) FIG. **24**B illustrates a cross-section side view of an integrated circuit package assembly **2470**. The integrated circuit package assembly **2470** illustrates an implementation of one or more processor or accelerator devices as described herein. The package assembly **2470** includes multiple units of hardware logic **2472**, **2474** connected to a substrate **2480**. The logic **2472**, **2474** may be implemented at least partly in configurable logic or fixed-functionality logic hardware, and can include one or more portions of any of the processor core(s), graphics processor(s), or other accelerator devices described herein. Each unit of logic **2472**, **2474** can be implemented within a semiconductor die and coupled with the substrate **2480** via an interconnect structure **2473**. The interconnect structure **2473** may be configured to route electrical signals between the logic **2472**, **2474** and the substrate **2480**, and can include interconnects such as, but not limited to bumps or pillars. The interconnect structure **2473** may be configured to route electrical signals such as, for example, input/output (I/O) signals and/or power or ground signals associated with the operation of the logic **2472**, **2474**. Optionally, the substrate **2480** may be an epoxy-based laminate substrate. The substrate **2480** may also include other suitable types of substrates. The package assembly **2470** can be connected to other electrical devices via a package interconnect **2483**. The package interconnect **2483** may be coupled to a surface of the substrate **2480** to route electrical signals to other electrical devices, such as a motherboard, other chipset, or multi-chip module.

(348) The units of logic **2472**, **2474** may be electrically coupled with a bridge **2482** that is configured to route electrical signals between the logic **2472**, **2474**. The bridge **2482** may be a dense interconnect structure that provides a route for electrical signals. The bridge **2482** may include a bridge substrate composed of glass or a suitable semiconductor material. Electrical routing features can be formed on the bridge substrate to provide a chip-to-chip connection between the logic **2472**, **2474**.

(349) Although two units of logic **2472**, **2474** and a bridge **2482** are illustrated, embodiments described herein may include more or fewer logic units on one or more dies. The one or more dies may be connected by zero or more bridges, as the bridge **2482** may be excluded when the logic is included on a single die. Alternatively, multiple dies or units of logic can be connected by one or more bridges. Additionally, multiple logic units, dies, and bridges can be connected together in other possible configurations, including three-dimensional configurations.

(350) FIG. **24**C illustrates a package assembly **2490** that includes multiple units of hardware logic chiplets connected to a substrate **2480** (e.g., base die). A graphics processing unit, parallel processor, and/or compute accelerator as described herein can be composed from diverse silicon chiplets that are separately manufactured. In this context, a chiplet is an at least partially packaged integrated circuit that includes distinct units of logic that can be assembled with other chiplets into a larger package. A diverse set of chiplets with different IP core logic can be assembled into a single device. Additionally the chiplets can be integrated into a base die or base chiplet using active interposer technology. The concepts described herein enable the interconnection and communication between the different forms of IP within the GPU. IP cores can be manufactured using different process technologies and composed during manufacturing, which avoids the complexity of converging multiple IPs, especially on a large SoC with several flavors IPs, to the same manufacturing process. Enabling the use of multiple process technologies improves the time to market and provides a cost-effective way to create multiple product SKUs. Additionally, the disaggregated IPs are more amenable to being power gated independently, components that are not in use on a given workload can be powered off, reducing overall power consumption.
(351) The hardware logic chiplets can include special purpose hardware logic chiplets **2472**, logic or I/O chiplets **2474**, and/or memory chiplets **2475**. The hardware logic chiplets **2472** and logic or I/O chiplets **2474** may be implemented at least partly in configurable logic or fixed-functionality logic hardware and can include one or more portions of any of the processor core(s), graphics processor(s), parallel processors, or other accelerator devices described herein. The memory chiplets **2475** can be DRAM (e.g., GDDR, HBM) memory or cache (SRAM) memory.
(352) Each chiplet can be fabricated as separate semiconductor die and coupled with the substrate **2480** via an interconnect structure **2473**. The interconnect structure **2473** may be configured to route electrical signals between the various chiplets and logic within the substrate **2480**. The interconnect structure **2473** can include interconnects such as, but not limited to bumps or pillars. In some embodiments, the interconnect structure **2473** may be configured to route electrical signals such as, for example, input/output (I/O) signals and/or power or ground signals associated with the operation of the logic, I/O and memory chiplets.
(353) The substrate **2480** may be an epoxy-based laminate substrate, however, it is not limited to that and the substrate **2480** may also include other suitable types of substrates. The package assembly **2490** can be connected to other electrical devices via a package interconnect **2483**. The package interconnect **2483** may be coupled to a surface of the substrate **2480** to route electrical signals to other electrical devices, such as a motherboard, other chipset, or multi-chip module.
(354) A logic or I/O chiplet **2474** and a memory chiplet **2475** may be electrically coupled via a bridge **2487** that is configured to route electrical signals between the logic or I/O chiplet **2474** and a memory chiplet **2475**. The bridge **2487** may be a dense interconnect structure that provides a route for electrical signals. The bridge **2487** may include a bridge substrate composed of glass or a suitable semiconductor material. Electrical routing features can be formed on the bridge substrate to provide a chip-to-chip connection between the logic or I/O chiplet **2474** and a memory chiplet **2475**. The bridge **2487** may also be referred to as a silicon bridge or an interconnect bridge. For example, the bridge **2487** is an Embedded Multi-die Interconnect Bridge (EMIB). Alternatively, the bridge **2487** may simply be a direct connection from one chiplet to another chiplet.
(355) The substrate **2480** can include hardware components for I/O **2491**, cache memory **2492**, and other hardware logic **2493**. A fabric **2485** can be embedded in the substrate **2480** to enable communication between the various logic chiplets and the logic **2491**, **2493** within the substrate **2480**. Optionally, the I/O **2491**, fabric **2485**, cache, bridge, and other hardware logic **2493** can be integrated into a base die that is layered on top of the substrate **2480**. The fabric **2485** may be a network on a chip interconnect or another form of packet switched fabric that switches data packets between components of the package assembly.
(356) Furthermore, a package assembly **2490** can also include a smaller or greater number of

components and chiplets that are interconnected by a fabric **2485** or one or more bridges **2487**. The chiplets within the package assembly **2490** may be arranged in a 3D or 2.5D arrangement. In general, bridge structures **2487** may be used to facilitate a point to point interconnect between, for example, logic or I/O chiplets and memory chiplets. The fabric **2485** can be used to interconnect the various logic and/or I/O chiplets (e.g., chiplets **2472**, **2474**, **2491**, **2493**). with other logic and/or I/O chiplets. The cache memory **2492** within the substrate can act as a global cache for the package assembly **2490**, part of a distributed global cache, or as a dedicated cache for the fabric **2485**.

(357) FIG. **24**D illustrates a package assembly **2494** including interchangeable chiplets **2495**, according to an embodiment. The interchangeable chiplets **2495** can be assembled into standardized slots on one or more base chiplets **2496**, **2498**. The base chiplets **2496**, **2498** can be coupled via a bridge interconnect **2497**, which can be similar to the other bridge interconnects described herein and may be, for example, an EMIB. Memory chiplets can also be connected to logic or I/O chiplets via a bridge interconnect. I/O and logic chiplets can communicate via an interconnect fabric. The base chiplets can each support one or more slots in a standardized format for one of logic or I/O or memory/cache.

(358) SRAM and power delivery circuits may be fabricated into one or more of the base chiplets **2496**, **2498**, which can be fabricated using a different process technology relative to the interchangeable chiplets **2495** that are stacked on top of the base chiplets. For example, the base chiplets **2496**, **2498** can be fabricated using a larger process technology, while the interchangeable chiplets can be manufactured using a smaller process technology. One or more of the interchangeable chiplets **2495** may be memory (e.g., DRAM) chiplets. Different memory densities can be selected for the package assembly **2494** based on the power, and/or performance targeted for the product that uses the package assembly **2494**. Additionally, logic chiplets with a different number of type of functional units can be selected at time of assembly based on the power, and/or performance targeted for the product. Additionally, chiplets containing IP logic cores of differing types can be inserted into the interchangeable chiplet slots, enabling hybrid processor designs that can mix and match different technology IP blocks.

(359) Exemplary System on a Chip Integrated Circuit

(360) FIG. **25** and FIG. **26**A-**26**B illustrate exemplary integrated circuits and associated graphics processors that may be fabricated using one or more IP cores. In addition to what is illustrated, other logic and circuits may be included, including additional graphics processors/cores, peripheral interface controllers, or general-purpose processor cores. The elements of FIG. **25** and FIG. **26**A-**26**B having the same or similar names as the elements of any other figure herein describe the same elements as in the other figures, can operate or function in a manner similar to that, can comprise the same components, and can be linked to other entities, as those described elsewhere herein, but are not limited to such.

(361) FIG. **25** is a block diagram illustrating an exemplary system on a chip integrated circuit **2500** that may be fabricated using one or more IP cores. Exemplary integrated circuit **2500** includes one or more application processor(s) **2505** (e.g., CPUs), at least one graphics processor **2510**, which may be a variant of the graphics processor **1408**, **1508**, **2510**, or of any graphics processor described herein and may be used in place of any graphics processor described. Therefore, the disclosure of any features in combination with a graphics processor herein also discloses a corresponding combination with the graphics processor **2510**, but is not limited to such. The integrated circuit **2500** may additionally include an image processor **2515** and/or a video processor **2520**, any of which may be a modular IP core from the same or multiple different design facilities. Integrated circuit **2500** may include peripheral or bus logic including a USB controller **2525**, UART controller **2530**, an SPI/SDIO controller **2535**, and an I.sup.2S/I.sup.2C controller **2540**. Additionally, the integrated circuit can include a display device **2545** coupled to one or more of a high-definition multimedia interface (HDMI) controller **2550** and a mobile industry processor interface (MIPI) display interface **2555**. Storage may be provided by a flash memory subsystem

2560 including flash memory and a flash memory controller. Memory interface may be provided via a memory controller **2565** for access to SDRAM or SRAM memory devices. Some integrated circuits additionally include an embedded security engine **2570**.
(362) FIG. **26**A-**26**B are block diagrams illustrating exemplary graphics processors for use within an SoC, according to embodiments described herein. The graphics processors may be variants of the graphics processor **1408**, **1508**, **2510**, or any other graphics processor described herein. The graphics processors may be used in place of the graphics processor **1408**, **1508**, **2510**, or any other of the graphics processors described herein. Therefore, the disclosure of any features in combination with the graphics processor **1408**, **1508**, **2510**, or any other of the graphics processors described herein also discloses a corresponding combination with the graphics processors of FIG. **26**A-**26**B, but is not limited to such. FIG. **26**A illustrates an exemplary graphics processor **2610** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. FIG. **26**B illustrates an additional exemplary graphics processor **2640** of a system on a chip integrated circuit that may be fabricated using one or more IP cores, according to an embodiment. Graphics processor **2610** of FIG. **26**A is an example of a low power graphics processor core. Graphics processor **2640** of FIG. **26**B is an example of a higher performance graphics processor core. For example, each of the graphics processors **2610**, **2640** can be a variant of the graphics processor **2510** of FIG. **25**, as mentioned at the outset of this paragraph.
(363) As shown in FIG. **26**A, graphics processor **2610** includes a vertex processor **2605** and one or more fragment processor(s) **2615**A-**2615**N (e.g., **2615**A, **2615**B, **2615**C, **2615**D, through **2615**N-**1**, and **2615**N). Graphics processor **2610** can execute different shader programs via separate logic, such that the vertex processor **2605** is optimized to execute operations for vertex shader programs, while the one or more fragment processor(s) **2615**A-**2615**N execute fragment (e.g., pixel) shading operations for fragment or pixel shader programs. The vertex processor **2605** performs the vertex processing stage of the 3D graphics pipeline and generates primitives and vertex data. The fragment processor(s) **2615**A-**2615**N use the primitive and vertex data generated by the vertex processor **2605** to produce a framebuffer that is displayed on a display device. The fragment processor(s) **2615**A-**2615**N may be optimized to execute fragment shader programs as provided for in the OpenGL API, which may be used to perform similar operations as a pixel shader program as provided for in the Direct 3D API.
(364) Graphics processor **2610** additionally includes one or more memory management units (MMUs) **2620**A-**2620**B, cache(s) **2625**A-**2625**B, and circuit interconnect(s) **2630**A-**2630**B. The one or more MMU(s) **2620**A-**2620**B provide for virtual to physical address mapping for the graphics processor **2610**, including for the vertex processor **2605** and/or fragment processor(s) **2615**A-**2615**N, which may reference vertex or image/texture data stored in memory, in addition to vertex or image/texture data stored in the one or more cache(s) **2625**A-**2625**B. The one or more MMU(s) **2620**A-**2620**B may be synchronized with other MMUs within the system, including one or more MMUs associated with the one or more application processor(s) **2505**, image processor **2515**, and/or video processor **2520** of FIG. **25**, such that each processor **2505**-**2520** can participate in a shared or unified virtual memory system in which processors within the system share a single virtual address space. Components of graphics processor **2610** may correspond with components of other graphics processors described herein. The one or more MMU(s) **2620**A-**2620**B may correspond with MMU **245** of FIG. **2**C. Vertex processor **2605** and fragment processor(s) **2615**A-**2615**N may correspond with graphics multiprocessor **234**. The one or more circuit interconnect(s) **2630**A-**2630**B enable graphics processor **2610** to interface with other IP cores within the SoC, either via an internal bus of the SoC or via a direct connection, according to embodiments. The one or more circuit interconnect(s) **2630**A-**2630**B may correspond with the data crossbar **240** of FIG. **2**C. Further correspondence may be found between analogous components of the graphics processor **2610** and the various graphics processor architectures described herein.
(365) As shown FIG. **26**B, graphics processor **2640** includes the one or more MMU(s) **2620**A-

2620B, cache(s) **2625**A-**2625**B, and circuit interconnect(s) **2630**A-**2630**B of the graphics processor **2610** of FIG. **26**A. Graphics processor **2640** includes one or more shader cores **2655**A-**2655**N (e.g., **2655**A, **2655**B, **2655**C, **2655**D, **2655**E, **2655**F, through **2655**N-**1**, and **2655**N), which provides for a unified shader core architecture in which a single core or type or core can execute all types of programmable shader code, including shader program code to implement vertex shaders, fragment shaders, and/or compute shaders. The exact number of shader cores present can vary among embodiments and implementations. Additionally, graphics processor **2640** includes an inter-core task manager **2645**, which acts as a thread dispatcher to dispatch execution threads to one or more shader cores **2655**A-**2655**N and a tiling unit **2658** to accelerate tiling operations for tile-based rendering, in which rendering operations for a scene are subdivided in image space, for example to exploit local spatial coherence within a scene or to optimize use of internal caches. Shader cores **2655**A-**2655**N may correspond with, for example, graphics multiprocessor **234** as in FIG. **2**D, or graphics multiprocessors **325**, **350** of FIGS. **3**A and **3**B respectively, or multi-core group **365**A of FIG. **3**C.

(366) The graphics processing technologies described above may include technologies, systems, methods, and techniques to enable the dynamic reconfiguration of memory on a GPGPU. One embodiment described herein enables dynamic reconfiguration of cache memory bank assignments based on hardware statistics. One embodiment enables for virtual memory address translation using mixed four kilobyte and 64 kilobyte pages within the same page table hierarchy and under the same page directory. One embodiment provides for a GPGPU and associated heterogenous processing system having near and far regions of the same level of a cache hierarchy.

(367) Dynamic Reconfiguration of Cache Memory Bank Assignments

(368) Cache memory bank assignment for L1, L2, and L3 caches is performed based on a static hash, which may lead to hot spotting, resulting in performance issues. A hardware monitor that detects bank hot spotting can be used to re-calculate a new hash to enable bank assignments to change. When a hash change is to occur within a cache, first the cache is flushed, then the hash used to determine bank assignments is changed. The cache may then be re-loaded with previously stored data or the cache can be allowed to fill normally. In one embodiment, memory bank hash changes can piggyback on an existing cache flush.

(369) FIG. **27** illustrates a processing system **2700** including a banked cache memory **2708**. The processing system **2700** includes a processing resource **2702**, a load/store unit **2704**, a cache controller **2706**, the cache memory **2708**, and memory **2720**. The processing system **2700** illustrates a technique in which data that is stored in the cache memory **2708** in conjunction with a read from or write to the memory **2720** is assigned to one or more memory banks **2710**A-**2710**N based on hash logic **2707**. The one or more memory banks **2710**A-**2710**N make up the physical storage medium of the cache memory **2708**. The hash logic **2707** can apply one or more hash functions to an address associated with a memory access to determine a bank **2710**A-**2710**N in which to store one or more cache lines of data.

(370) The cache memory **2708** can be any cache memory described herein, such as but not limited to L1 cache **248**, **1554**, L2 cache **1553**, cache memories **272**, **342**, **358**A-**358**B, **438**, **708**, **1404**, **1504**A-**1504**N, **1725**, **2492**, data cache **1812**, L3 cache **2175**, or any other cache described herein. The cache memory **2708** may also be combination cache/shared memory, such as but not limited to L1 cache/shared memory **373** and/or shared memory/cache memory **1536**.

(371) The processing resource **2702** represents a processing element (e.g., GPGPU core, ray-tracing core, tensor core, execution resource, execution unit (EU), stream processor, streaming multiprocessor (SM), graphics multiprocessor) associated with a graphics processor or graphics processor structure (e.g., parallel processing unit, graphics processing engine, multi-core group, compute unit, compute unit of graphics core next) in a GPU as described herein. For example, the processing resource **2702** may be one of the GPGPU cores **262**, or tensor/ray-tracing cores **263** of graphics multiprocessor **234**; a ray-tracing core **338**A-**338**B, tensor core **337**A-**337**B or GPGPU

core **336**A-**336**B of graphics multiprocessor **325**; execution resources **356**A-**356**D of graphics multiprocessor **350**; one of GFX cores **370**, tensor cores **371**, or ray tracing cores **372** of a multi-core group **365**A-**365**N; one of vector logic units **1563** or scalar logic units **1564** of a compute unit **1506**A-**1506**N; execution unit with EU array **1522**A-**1522**F or EU array **1524**A-**1524**F; an execution unit **1808**A-**1808**N of execution logic **1800**; and/or execution unit **1900**. The processing resource **2702** may also be an execution resource within, for example, a graphics processing engine **431-432**, GPGPU hardware **610**, GPGPU **700**, processing cluster **706**A-**706**H, GPGPU **806**A-**806**D, GPGPU **1306**, graphics processing engine **1610**, graphics processing engine cluster **1622**, and/or graphics processing engine **1710**. The processing resource **2702** may also be a processing resource within graphics processor **2510**, graphics processor **2610**, and/or graphics processor **2640**.

(372) The load/store unit **2704** can be any load/store unit described herein, such as but not limited to load/store unit **266**, load/store unit **340**A-**340**B. The load/store unit **2704** facilitate access to memory for the processing resource **2702**. Accesses to memory may be cached or un-cached accesses. Cached memory accesses traverse the cache memory **2708**, while un-cached memory access may be made directly to the memory **2720**, without using the cache memory **2708** to store data written to memory **2720** or read from memory **2720**.

(373) The cache controller **2706** manages cache lines in the cache memory **2708** and copies data to and from the cache based on cached memory accesses. The cache controller **2706** can intercept read and write memory requests before passing the requests to the memory controller associated with the memory **2720**. The cache controller **2706** can also manage operations associated with the cache coherency protocol in which the cache memory **2708** participates.

(374) In various cache configurations, and depending on the size of the memory banks **2710**A-**2710**N, a unit of cached data (e.g., cache line) may be stored within a single one of the memory banks **2710**A-**2710**N or can span multiple banks. The cache controller **2706** can include hash logic **2707** to apply an address hashing function may be used to determine where a given piece of data is to be stored within the cache memory **2708**. For example, in one configuration, based on corresponding memory address to be access, data associated with the memory access may be stored in one or more memory banks **2710**A-**2710**N based on output of the hashing function used by the hash logic **2707**. Where data associated with a memory access may be stored within a single bank, the hash function may output a value between 0 and N, where each potential value between 0 and N may correspond to one of the memory banks **2710**A-**2710**N. Where data associated with a memory access is to span multiple memory banks **2710**A-**2710**N, the hash logic **2707** can select multiple banks multiple memory banks **2710**A-**2710**N in which to store the data.

(375) Where instructions executed by the processing resource **2702** repeat the same memory access pattern, the same bank or set of banks may be frequently accessed by such instructions. These frequent accesses may result in hotspots developing within certain memory banks **2710**A-**2710**N. The hotspots may induce negative thermal effects into the processing system **2700**, reduce the throughput associated certain memory accesses, and/or increase the latency associated with those memory accesses.

(376) FIG. **28** illustrates a processing system **2800** including a cache controller **2806** with dynamic hash unit **2807**. The processing system **2800** additionally includes cache monitor hardware **2810**. The cache monitor hardware **2810** can be included within the cache controller **2806**. The cache monitor hardware **2810** may also be located externally to the cache controller **2806**. For example, cache monitor hardware **2810** may reside within the cache memory **2708**. The dynamic hash unit **2807** may be hardware logic within the cache controller **2806**. The dynamic hash unit **2807** may also be updatable firmware that includes instructions executed by a microcontroller within or associated with the cache controller **2806**. The cache memory **2708** may be the same or similar memory as in processing system **2700** and includes multiple memory banks **2710**A-**2710**N. The processing system **2800** also includes an additional cache or memory **2816**, which can be an additional cache memory, which may be a higher-level cache memory, or main system memory,

such as memory **2720**.

(377) In one embodiment the dynamic hash unit **2807** is programmed with awareness of the specific memory access patterns performed by the processing resource **2702**. For example, the processing resource **2702** may be configured with a tiled memory access pattern having a pre-defined pitch and stride. The hash algorithms used by the dynamic hash unit **2807** may be adapted for the specific type of memory access patterns employed by the processing resource **2702**. Having knowledge of the memory access pattern enables the dynamic hash unit **2807** to select between different hashing algorithms that will be known to select different sets of memory banks **2710**A-**2710**N when the same memory access pattern is used.

(378) During execution, the cache monitor hardware **2810** can monitor the bank access pattern for the cache memory **2708**. For example, the cache monitor hardware **2810** can include a counter array **2811** that updates a count for each memory bank **2710**A-**2710**N when an access to that memory bank is performed. The count in the counter array **2811** can be a count of accesses over a sliding window of time. The counter array **2811** may also be periodically reset. For example, the counter array **2811** may be reset in response to a cache flush. The counter array **2811** may also be reset in response to a context switch at the processing resource **2702**. Via the counter array **2811**, the cache monitor hardware **2810** can detect whether specific memory banks **2710**A-**2710**N are receiving a disproportionate number of accesses relative to other memory banks. If the cache monitor hardware **2810** detects a disproportionate number of accesses to specific memory banks **2710**A-**2710**N relative to other memory banks, such that an access differential is over a pre-determined threshold, the cache monitor hardware **2810** can direct the cache controller **2806** to perform an operation **2812** to reprogram the dynamic hash unit **2807**.

(379) In one configuration, each memory bank **2710**A-**2710**N in the cache memory **2708** may be configured with a thermal sensor. A thermal monitor **2818** can be configured to monitor the temperature each memory bank **2710**A-**2710**N. In such configuration, the thermal monitor **2818** can signal the cache monitor hardware **2810** when the temperature of a memory bank exceeds a threshold. The cache monitor hardware **2810** can then perform the operation **2812** to reprogram the dynamic hash unit **2807** in response to the signal from the thermal monitor **2818**.

(380) The operation **2812** to reprogram the dynamic hash unit **2807** can include requesting the dynamic hash unit **2807** to switch to a different hash function that will result in a different bank access pattern. The dynamic hash unit **2807** can then switch to a different one of multiple hash functions that are available within the hardware or firmware of the dynamic hash unit **2807**. In one embodiment, the dynamic hash unit **2807** may be configured to accept an updated hash function that may override one of the pre-configured hash functions of the dynamic hash unit **2807**. This updated hash function may be added to the set of available hash functions that may be selected by operation **2812**.

(381) When a change in hash function is performed by the dynamic hash unit **2807**, a flush of the cache memory **2708** is triggered. During the flush, any modified (e.g., dirty) units (e.g., cache lines) of cache data may be flushed to the additional cache or memory **2816**. Unmodified units of cache data may be discarded or invalidated. In one embodiment, all cache lines in the cache may be set to invalid before changing a hash function used for bank selection. After the cache memory **2708** is flushed, the dynamic hash unit **2807** can select a different hash function that will result in a different bank access pattern. The cache controller **2806** can then resume operation of the cache memory **2708**. The dynamic hash unit **2807** can also piggyback hash function changes on a cache flush requested by the processing resource **2702**.

(382) FIG. **29** illustrates a method **2900** to enable dynamic reconfiguration of a memory bank hash based on hardware statistics. The method **2900** can be performed by cache monitor hardware **2810** based on hardware statistics for cache memory bank accesses and/or thermal states. The method **2900** includes to perform operations (**2902**) by the cache monitor hardware to monitor access pattern and/or thermal state of memory banks of a cache memory. If the cache monitor hardware

detects a bank access disparity over a threshold (**2903**) or a bank temperature over a threshold (**2904**), the method **2900** indicates for the cache monitor hardware to request a hash function change at the dynamic hash unit (**2906**). Otherwise, the cache monitor hardware can continue to perform the operations (**2902**) to monitor the access pattern and/or thermal state of the memory banks of the cache memory.

(383) In one configuration, the request for the hash function change can request to rotate to a different hash function or can be a request to select a specific hash function. The hash function that is selected may differ based on whether the change is based on a thermal state or an access pattern disparity.

(384) Using the techniques above, one skilled in the art may implement, for example, a data processing comprising a memory and a general-purpose graphics processor coupled with the memory. The general-purpose graphics processor includes a cache memory including multiple memory banks and a cache controller coupled with the cache memory. The cache memory may be a level 1 (L1), level 2 (L2), or level 3 (L3) cache memory of the general-purpose graphics processor.

(385) The cache controller may include a dynamic hash unit to select, based on a first hash function and an address associated with a memory access, one or more of the multiple memory bank to which data associated with the memory access is to be stored. The system also includes cache monitor hardware to monitor an access pattern of the multiple memory banks. In response to a detected access pattern, the cache monitor hardware can request for the dynamic hash unit to select a second hash function in response to detection of a bank access pattern disparity over a threshold.

(386) In one embodiment the cache memory additionally includes thermal monitoring hardware to monitor a thermal state of the multiple memory banks and report the thermal state to the cache monitor hardware. The cache monitor hardware can receive data from the thermal monitoring hardware and request the dynamic hash unit to select the second hash function in response to a determination that a temperature of one or more of the multiple memory banks exceeds a threshold. The cache monitor can request that the dynamic hash unit selects a third hash function in response to a determination that a temperature of one or more of the multiple memory banks exceeds a threshold.

(387) In one embodiment the dynamic hash unit is a hardware unit of the cache controller. The cache controller include, be included within, or comprise firmware executed by a microcontroller. The dynamic hash unit may include or be included within firmware executed by such microcontroller.

(388) Systems and methods may be implemented to manage the above features or that may include aspects of the above features. Non-transitory machine readable media may store instructions that cause processors and/or microcontrollers to provide the above mentioned features.

(389) Mixed 4K/64K Pages Under the Same Page Directory

(390) State of the art computing systems in common general use include 64-bit processors that support up to 52 bit physical addresses and employ a paged virtual memory system with up to 48-bit virtual addresses. Paged virtual memory systems used by such processors may employ a variety of hierarchical page table structures to managed virtual to physical address mapping. A virtual address can specify a virtual page number. The virtual page number may be translated by an address translation system into a physical page number that identifies a page in physical memory. The virtual address also specifies an offset into the page that specifies a distance from the page base address in which data the specified address may be found.

(391) Heterogenous processing systems that enable cooperative processing between multiple types of processors, such as a general purpose processor (e.g., CPU or Application Processor) and a general purpose graphics processor unit (GPGPU) may employ unified memory in a single view of the system virtual address space is shared by all processors in the system. A GPGPU within a heterogenous processing system that includes support for unified memory includes hardware and software that enables the use virtual memory allocations that are created for use by the GPGPU, as

well as allocations that are created at least partially for use by a CPU and/or other processors within the system. Both CPUs and GPGPUs can support a variety of page sizes for virtual memory allocations, from four kilobyte (4K) and sixty-four kilobyte (64K) pages, up to through one gigabyte (1G) and/or sixteen gigabyte (16G) pages. Generally, 4K is the common and/or default page size for a variety of CPU architectures, with both 4K and 64K being common for use by GPGPUs.

(392) FIG. **30** illustrates a heterogenous processing system **3000** including unified memory **3040**. The heterogenous processing system **3000** includes a CPU **3002**, system memory **3004**, GPGPU memory **3006**, and a GPGPU **3008**. The CPU **3002** may be any CPU, general-purpose processor, or application processor described herein, such as, for example, processor **102**, processor **802**, multi-core processor **1308**, processor(s) **1402**, processor **2330**, and/or application processor **2505**. The system memory **3004** is random access memory and may be, for example, system memory **104**, memory **366**, processor memory **401-402**, system memory **441**, memory **1571**, memory **2350**, and/or comprise one or more of memory device **1420**. The GPGPU memory **3006** may be a portion of system memory **3004** that is dedicated for use by the GPGPU **3008**. The GPGPU memory **3006** may also be local, dedicated, or-board memory, such as parallel processor memory **222**, GPU memory **420-423**, GFX memory **433-434**, memory **714**A-**714**B, memory **1572**, memory **1626**A-**1626**D, and/or memory **2475**. The GPGPU may be any graphics or parallel processor described herein, such as but not limited to parallel processor(s) **112**, GPU **380**, GPU **410-413**, graphics acceleration module **446**, GPGPU hardware **610**, GPGPU **700**, GPGPU **806**A-**806**D, GPGPU **1306**, GPGPU **1570**, graphics processor **1620**, compute accelerator **1630**, graphics processor **1710**, graphics processor **2100**, graphics processor **2510**, graphics processor **2610**, and/or graphics processor **2640**. Techniques described herein may also apply to image processor **2515** and video processor **2520**. The system memory **3004** and GPGPU memory **3006** may have unified or non-unified physical address spaces and are configured as unified memory **3040** with a unified virtual address space, such as, for example, the unified memory of FIG. **4**F.

(393) The CPU **3002** and GPGPU **3008** each includes MMUs **3003**, **3028**, which are similar to other memory management units described herein, to facilitate access to the system memory **3004** and GPGPU memory **3006**. MMU **3028** includes a TLB **3032** to cache virtual to physical address translations, and a page table walker **3030** that includes hardware to walk the GPU page tables **3016** and/or CPU page tables **3024** upon a TLB miss. CPU MMU **3003** may also include similar components. Where the CPU **3002** is a multi-core CPU, each core may include a separate instance of MMU **3003**.

(394) In various implementations the CPU page tables **3024** and GPU page tables **3016** may be synchronized or the CPU **3002** and GPGPU **3008** may be configured to share a unified set of page tables that encompasses the data of the CPU page tables **3024** and the GPU page tables **3016**. During operation, a graphics driver (e.g., GPGPU driver **608**, user mode graphics driver **2326**, kernel mode graphics driver **2329**) that executes on the CPU **3002**, based on commands received via a graphics or compute API, can load commands into a command buffer **3014** in system memory **3004**. Addresses in the system memory **3004** that store the command buffer **3014** can be mapped via the GPU page table **3016** to the GPGPU **3008** to enable a render engine **3018** of the GPGPU **3008** to fetch and execute those commands. Pixels generated by the render engine **3018** may be written to a frame buffer **3026** in GPGPU memory **3006** and read by a display engine **3038** for output to a display device.

(395) As the heterogeneous processing system **3000** includes unified memory **3040**, it would be advantageous to enable mappings for 4K virtual memory pages to exist alongside mappings for 64K pages. For example, the GPGPU **3008** can create 4K virtual memory pages for the use of the CPU **3002** or GPGPU **3008** alongside 64K pages created by or for the use of the GPGPU **3008**. For processing systems with support multiple page sizes, such systems may employ different hierarchical structures depending on the page size in use. If different hierarchical structures are not

used, the way those structures are created and used may differ depending on the page size in use based on the mathematics associated with the identification of a virtual page number and offset.

(396) FIG. **31**A-**31**B illustrates hierarchical page table structures for 4K and 64K pages. A system with support for 4K pages may use the page table structure of FIG. **31**A. A system with support for 64K pages may use the page table structure of FIG. **31**B. In each of FIG. **31**A and FIG. **31**B, a four level hierarchical page table structure is illustrated. The illustrated page table structures are exemplary of one embodiment and different techniques may be used to enable support for virtual memory systems using 4K or 64K page sizes. The illustrates page table structures are for use by 64-bit processors having at least a 48-bit canonical virtual address space. The illustrated page table structures may also be used by GPGPUs having a 49-bit virtual address space, where the 49-bit virtual address space enables the GPGPU to address the entirely of the address space available to a CPU having a 48-bit address space, as well as higher virtual addresses dedicated for use by the GPGPU.

(397) As shown in FIG. **31**A, a virtual address provides indices into a hierarchical page table structure that is used to enable virtual to physical address mapping for the virtual address. When creating virtual to physical address mappings for a 4K page, a 9:9:9:9 page table walk may be used in which four 9-bit regions of the virtual address are used to identify the page table that stores the virtual to physical mapping for the virtual address. Bits [47:39] of the virtual address specify page map level 4 (PML4) index **3102**. Address bits [38:30] specify a page directory pointer (PDP) index **3104**. Address bits [29:21] specify a page directory index **3106**. Address bits [20:12] specify a page table index **3108**. Once the virtual page to physical page translation is performed, address bits [11:0] are used to determine an offset-inside-page **3110** for data associated with the virtual address. In one implementation, a PML4 pointer **3111** that specifies the base address of a PML4 table **3112** may be found, for example, in a process address space ID (PASID) entry for a PCI device having support for the PCI PASID extension. In other implementations, similar pointers may be stored in registers (e.g., CR3), context descriptors, or the like.

(398) The page table walk beings using the PML4 entry (PML4E) specified by the PML4 index **3102**, which specifies the physical address of a page directory pointer (PDP) table **3114** within a plurality of possible PDP tables can be identified. The PDP table **3114** includes a page directory pointer entry (PDPE). The PDPE identifies the physical address of page directory table **3116**. The page directory index **3106** of the virtual address specifies a PDE (page directory entry) that includes the physical address of a page table **3118**. The page table index **3108** specifies a page table entry (PTE) the specifies the physical address of a 4K memory page **3120**, with the offset-inside-page **3110** specifying the offset at which data associated with the virtual address is stored. The 9:9:9:9 page table walk results in a fully utilized page table, such that a 4K memory page can be used to store a page table of 29 (512) page table entries, with each entry being eight bytes (64-bit), without any memory gaps between the entries.

(399) As shown in FIG. **31**B, however, when the page table hierarchy of FIG. **31**A is used for 64 KB pages, the memory used to store the page table is not fully utilized, resulting in unused space between page table entries. The page table hierarchy of FIG. **31**B is similar to that of FIG. **31**A, excepting that bits [20:16] of the virtual address are used as the page table index **3138** and bits [15:0] are used for the offset-inside-page **3140**. Sixteen bits of offset are used to cover the 64 kilobytes of the 64K page. Within the page table **3148**, every sixteenth entry (PTE #0, PTE #16, PTE #32, . . . PTE #496) is used to index a 64K page, which may be calculated using virtual address[20:16] & “0000”.

(400) FIG. **32**A-**32**C illustrate a page directory and page tables to enable the mixing of 4K and 64K pages within the same hierarchical page table structure, according to an embodiment. FIG. **32**A illustrates a page directory and page tables that can be used to index both 4K and 64K pages. FIG. **32**B illustrates fields of a page directory entry. FIG. **32**C illustrates fields of a page table entry.

(401) The illustrated page table hierarchy of FIG. **32**A-**32**C enhances TLB efficiency via the

addition of a 64K TLB hint to 4K PTEs. In one embodiment, a PS64 (page size 64) bit can be added to a 4K TLB that serves as a TLB coalescing hint. Coalescing enables address translation data for sixteen consecutive 4K PTEs to be cached in the TLB as 64 KB page. The memory arbitration and page table walking logic can continue to perform address translation for a coalesced 4K page as though the page is a 4K page. However, the coalesced 4K pages are cached as a 64K page. Caching the coalesced 4K pages as a 64K improves TLB efficiency by enabling an access to any of the sixteen coalesced PTEs to result in a hit to the same TLB entry.

(402) As shown in FIG. **32**A, the page table hierarchy includes a PDP table **3114**. The PDP table **3114** is 4K in size, aligned on a 4K address boundary, and may store up to 512 PDPEs. The PDPEs may reference a 1G page **3203** (e.g., PDPE **3201**) if the page structure (PS) field of the PDPE is set to “1”, or a PD table if PS=1 for the PDPE (e.g., PDPE **3202**). The page structure field may be a single-bit field.

(403) The page table hierarchy also includes a PD table **3116**. The PD table **3116** is 4K in size, is 4K aligned, and may store up to 512 PDEs. A PDE references a two-megabyte (2M) page **3207** if PS=1 for the PDE (e.g., PDE **3204**). In addition to the PS field, a PDE may also include a page table size (PTS) field that, in conjunction with the PS field, indicates whether the PDE references a PT32 page table **3148** or a PT512 page table **3118**. For example, PDE **3205** is set as {PS=0, PTS=1} and includes a physical address for a PT32 page table **3148**. The PT32 page table **3148** is 256 bytes (256B) in size, 256B aligned is used to store up to thirty-two 64K PTEs (e.g., 64K PTE **3208**) that include a physical address for a 64K page **3150**. As another example, PDE **3206** is set as {PS=0, PTS=1} and stores a physical address for a PT512 page table **3118**. The PT512 page table is 4K in size, aligned on a 4K address boundary, and may store up to 512 PTEs.

(404) A PTE in the PT512 page table **3118** can include the PS64 field, which may be a single bit field. The PS64 field is enabled (e.g., PS64=1) in sixteen consecutive 4K PTEs **3209** (e.g., PTE 0-15, 16-31, 32-47, etc.) when the corresponding 64 KB virtual address range maps to corresponding 4K chunks of a contiguous 64 KB page (16×4K pages **3220**) and all permission bits in the sixteen PTEs are the same. The sixteen consecutive non-coalesced 4K PTEs will have the PS64 field set to zero (e.g., PS64=0) and will each store the physical address **3210** for a single 4K memory page **3120**.

(405) FIG. **32**B shows exemplary fields of a PDE **3250** that enables mixed 4K and 64 pages. The PDE stores a page table address **3251** as well as indicator fields to indicate whether 64K pages are enabled on the system (field **3252**) and at PTS field **3254** that indicates whether the PDE is used to reference a page table that includes entries for 64K pages. The PTS field **3254** may be ignored if 64K pages are not enabled according to field **3252**. Field **3252** and PTS field **3254** may each be single bit fields, where a set bit for field **3252** indicates that 64K pages are enabled and a set bit for PTS field **3254** indicates that the PDE references a page table with support for entries that reference 64K pages. The PDE **3250** may also include a PS field **3256** that, when set, indicates that the PDE references a single 2M memory page.

(406) FIG. **32**C shows exemplary fields of a PTE **3260** that can be used to reference a 4K or a 64K page. The PTE stores a page address **3261**, which is the physical address of a 64K page. For a 4K page, an additional address field **3262** can be used to store additional bits that may be used for the physical address of a 4K page. The PS64 field **3264** indicates whether the page addressed by the PTE **3260** is to be cached in the TLB a single 4K page or as sixteen 4K pages that are coalesced into a 64K page.

(407) Table 5 below indicates the page size in a leaf entry and the corresponding mapping allowed in the TLB. An X entry indicates a “don't care” bit.

(408) TABLE-US-00005 TABLE 5 Page Table Hierarchy Leaf Entry Mapping Mapped TLB PS PTS PS64 Page Size Page Size 1 X X 2M 2M 0 1 X 64K 64K 0 0 0 4K 4K 0 0 1 4K 64K

(409) FIG. **33**A-**33**C illustrate methods **3300**, **3310**, **3320** to enable the mixing of 4K and 64K pages within a hierarchical page table structure. FIG. **33**A shows a method **3300** to create a

mapping for a page in a page table hierarchy that includes entries for 4K and 64K pages. FIG. **33**B shows a method **3310** of performing a page walk in a page table hierarchy that includes entries for 4K and 64K pages. FIG. **33**C shows a method **3320** of caching multiple 4K PTEs as a single 64K PTE. The illustrated methods **3300**, **3310** are applicable to a heterogenous processing system with mixed page sizes, for example, heterogenous processing system **3000** as in FIG. **30**.

(410) As shown in FIG. **33**A, logic on or associated with a GPGPU can receive a request to map to a 4K page associated with a CPU memory allocation (**3302**). The logic can create a mapping in a page table hierarchy associated with the GPGPU for the 4K page (**3303**). The logic can additionally receive a request on the GPGPU to map to a 64K page associated with a GPGPU memory allocation (**3304**). The logic can then create a mapping in the page table hierarchy associated with the GPGPU for the 64K page, where the page table hierarchy concurrently store entries for the 4K page and the 64K page (**3305**).

(411) As shown in FIG. **33**B, logic on or associated with a GPGPU can receive a request to translate a virtual address of a first memory allocation within a 4K page (**3312**). The translation can be performed via a TLB entry if the translation is cached within a TLB. In response to a TLB miss for the translation of the virtual address for the first memory allocation, the logic can initiate a page walk into a page table hierarchy to determine a physical address of the first memory allocation (**3313**). The page walk can be a 9:9:9:9 page table walk as described above. The logic can then receive a request to translate the virtual address of a second memory allocation within a 64K page (**3314**). The translation can be performed via a TLB entry if the translation is cached within a TLB. In response to a TLB miss for the translation of the virtual address for the second memory allocation, the logic can initiate a page walk into the page table hierarchy to determine the physical address for the second memory allocation (**3315**). The page table hierarchy stores a page table entry for the 4K page and the 64K page. In one embodiment a page table can store an entry for a 4K page and sixteen coalesced 4K PTEs that may be cached in a TLB as a 64K PTE.

(412) As shown in FIG. **33**C, logic on or associated with a GPGPU can receive a request to map to a 4K page for with a memory allocation associated with a CPU or a GPGPU (**3322**). The logic can map the 4K page using a 5K PTE within a 512 entry page table (**3323**). After the mapping or in parallel with the mapping of the 4K page, the logic can determine whether the mapped 4K page is within a 64K consecutive virtual address range (**3324**). If the mapped 4K page is within a 64K consecutive virtual address range, the logic can determine if the PTEs for each of the 4K pages within the consecutive virtual address range have the same permissions (**3325**). If the mapped 4K page is within a 64K consecutive virtual address range and the PTEs for each of the 4K pages within the consecutive virtual address range have the same permissions (e.g., read, write, execute, etc.), the logic can set a bit (e.g., PS64=1) in each of the sixteen PTEs for the consecutive 64K virtual address range (**3326**). The set bit indicates that the logic may cache the sixteen PTEs as a single 64K PTE (**3328**). Accordingly, an access to a virtual address that is covered by any of the sixteen PTEs will result in a TLB hit to the cached coalesced 64K PTE entry.

(413) Using the techniques above, one skilled in the art may implement, for example, an electronic device comprising a GPGPU including a memory management unit and a TLB that is configured to cache data associated with a translation from a virtual address to a physical address. The translation may be cached from a page table hierarchy configured to concurrently map a first page table entry for a 4K page table and a second page table entry associated with 64K page table. The second page table entry may include a bit that indicates that the second page table entry points to a 64K page. The second page table entry may be associated with sixteen 4K pages that are coalesced into a 64K page.

(414) In one embodiment the GPGPU of the electronic device includes a page table walker to walk a page table hierarchy that includes the page table upon a TLB miss for the translation from the virtual address to the physical address. The page table hierarchy includes a page directory table that includes a page directory entry. The page directory entry includes a reference to the page table. The

page directory entry may additionally include a or a multi-bit field that indicates that the page table referenced by the page table entry includes an entry for a 64K page or a 4K page. The page directory entry may also directly reference a 2M page. A 64K page reference by the page directory entry may be a monolithic 64K page or a set of coalesced 4K pages that are cached as a single 64K page. In one embodiment, an access to any one of the coalesced 4K pages will result in a hit to the same, single TLB entry.

(415) One skilled in the art would additionally be enabled to implement a method performed on an electronic device, where the method comprises receiving a request for GPGPU to map a 4K page associated with a first allocation, creating a mapping in a page table hierarchy associated with the GPGPU for the 4K page, receiving a request for the GPGPU to map to a sixty-four kilobyte (64K) page associated with a second allocation, and creating a mapping in the page table hierarchy associated with the GPGPU for the 64K page. The page table may concurrently store entries for the 4K page and the 64K page. The first memory allocation may be allocated for use at least in part by a central processing unit and is shared with the GPGPU. The second memory allocation may be allocated for use by the GPGPU. The method additionally includes, on the electronic device, determining that the 4K page is part of a contiguous 64K virtual address range and setting a bit in each page table entry of a set of page table entries for the contiguous 64K virtual address range. The set bit indicates that the set of page table entries for the 64K virtual address range is cacheable as a 64K page table entry. The method additionally includes caching the set of page table entries in a translation lookaside buffer as a 64K page table entry.

(416) One skilled in the art would additionally be enabled to implement a method performed on an electronic device, where the method comprises receiving a request at a GPGPU translate a virtual address of a first allocation within a four kilobyte (4K) page and in response to a TLB miss for translation of the virtual address for the first allocation, initiating a page walk into a page table hierarchy to determine a physical address for the first allocation. The method additionally includes receiving a request at the GPGPU translate the virtual address of a second allocation within a sixty-four kilobyte (64K) page and in response to a TLB miss for the translation of the virtual address for the second allocation, initiating a page walk into the page table hierarchy to determine a physical address for the second memory allocation. The electronic device is configured to include a page table hierarchy storing a page table entry for the 4K page and the 64K page. In one embodiment, the first memory allocation is allocated for use at least in part by a central processing unit and is shared with the GPGPU and the second memory allocation is allocated for use by the GPGPU.

(417) Near and Far Caches within a Single Layer of a Cache Hierarchy

(418) As the processing resources within a processor increases, it may be beneficial to increase the size of the caches used by those processing resources. Accordingly, cache memories, particularly a level-3 (L3), level-4 (L3) and/or last-level cache (LLC) for a processing system, tends to increase in size for successive processor designs within the same performance tier. While the throughput of these larger caches may be maintained, or even increased, as the cache size grows, access latency for a cache tends to increase as the size of the cache increase due to the management overhead of the larger number of cache entries. To counter the increase in latency for larger caches, a cache memory may be partitioned into a smaller near region and a larger far region. The smaller cache region can be used to store more latency sensitive and/or more frequently used data. Less latency sensitive or less frequently used data may be stored in the far cache region.

(419) In some systems, latency may be introduced due to a longer physical distance between a processor core and a large cache memory, or the nature of the communication channel between the processor core and the large cache memory. In such systems, latency may be reduced by including a smaller cache that is physically closer to the processor core or that is connected via a lower latency communication channel relative to the larger cache memory. For example, an L3/L4 or LLC for a processing system can be partitioned into a near and far region, with the near region having lower access latency.

(420) The near and far regions of a cache are treated as different regions of the same cache level, such that system-wide caching policies for a cache level apply to both the near and far region. For example, if a system has a multi-region L3 cache and a memory allocation is marked as not L3 cacheable, data for the memory allocation will not be stored in either region of the L3 cache.
(421) Cache eviction, replacement, and/or migration policies may be put in place to determine how victims are to be selected for eviction from each region of a multi-region cache and/or how or whether data is to be migrated between regions. Different algorithms may be used for the separate cache regions. A cache migration policies may be used to determine when or if data should be moved between higher latency and lower latency regions of the cache. Hints provided by instructions executed on a processor can also be used to determine which region should be used for initial data storage of an instruction and how resistant the data should be to migration or eviction. For example, a shader compiler that compiles a shader kernel for execution by a processing system can output compiler generated hints based on the detected memory access patterns for the compiled instructions. The instruction set for the shaders may also be extended to enable a programmer to include explicit hints for memory allocations. Those explicit hints can be used by a cache controller to determine a region in an L3/L4/LLC in which data for the kernel, or specific instructions in the kernel, should be cached.
(422) While near and far cache regions are advantageous for large high level caches, the techniques described herein are not limited to any particular layer of a cache hierarchy and are applicable to any level of a cache hierarchy.
(423) FIG. **34** illustrates a processing system **3400** including a cache **3408** having multiple regions. The cache **3408** may be a large L3, L4 or LLC cache, with a relatively smaller but lower latency near region **3418** and a relatively larger but higher latency far region **3428**. The processing system **3400** includes a processing resource **3402**, load/store unit **3404**, cache controller **3406**, cache **3408**, and memory **3420**. The elements of processing system **3400** are analogous to and may have characteristics similar to those of processing system **2700** of FIG. **27**. For example, processing resource **3402** and load/store unit **3404** may be similar to processing resource **2702** and load/store unit **2704**. Likewise, memory **3420** may be similar to memory **2720**. Cache **3408** may also be a banked cache, with separate memory banks for the near region **3418** and the far region **3428**.
(424) In addition to any bank selection logic that may be present for each region of the cache **3408**, the cache controller **3406** additionally includes cache management logic, such as region selection logic **3407**, one or more counter arrays **3417**, and/or cache replacement/migration logic **3427**. The counter arrays **3417** can be used to count a number of accesses to each unit of data (e.g., cache line) of each region within a period of time. When the need arises to evict data from a region, the cache replacement/migration logic **3427** may apply one or more migration policies to determine if a given cache line should be invalidated or, if dirty, written back to memory or a higher level cache. Various cache replacement algorithms may be used by the cache replacement/migration logic **3427** to determine to replace or migrate data based on usage metrics associated with the data. The cache replacement/migration logic **3427** can employ algorithms such as, but not limited to, least recently used (LRU), least frequently used (LFU), least frequently recently used (LFRU), or the like. Different replacement algorithms may be used for the near region **3418** and the far region **3428**. In one embodiment, both the near region **3418** and far region **3428** may be managed using a segmented least recently used (SLRU) algorithm, with the different segments of the SLRU algorithm being mapped to the different regions of the cache **3408**. The cache replacement/migration logic **3427** may be aware of and factor specific memory access patterns used by the processing resource **3402** when determining whether to replace or migrate a cache line. For example, if the processing resource **3402** is configured for tiled memory access, this access pattern may be used to predict cache lines that are likely to be accessed based on previously accessed cache lines.
(425) Using the SLRU algorithm or a dedicated cache migration algorithm, data may be migrated

between regions. Regularly accessed data in the far region **3428** may be migrated to the near region **3418**. Data that is evicted from the near region **3418** may be stored in the far region **3428** instead of being discarded from the cache.

(426) In one embodiment the cache management logic can store metadata to track the set of tags that are stored in the near region **3418** and far region **3428** of the cache **3408**. Alternatively, the cache management logic can check the tags in each region in parallel.

(427) FIG. **35** illustrates heterogenous processing system having near and far cache regions. The heterogenous processing system illustrates versions of the multi-core processor **407** and graphics acceleration module **446** of FIG. **4**C in which both the multi-core processor **407** and graphics acceleration module **446** include multi-region caches. In one embodiment, the caches (e.g., **462**A, **462**B, **462**C, **438** used by multi-core processor **407** are partitioned with near and far regions for a particular cache. For example, cache **462**A includes far region **462**A-F and near region **462**A-N. Cache **462**B includes far region **462**B-F and near region **462**B-N. Cache **462**B includes far region **462**B-F and near region **462**B-N. Cache **462**C includes far region **462**C-F and near region **462**C-N. Cache **462**D includes far region **462**D-F and near region **462**D-N. In one embodiment, cache **438** of the accelerator integration circuit **436** includes a far region **438**-F and a near region **438**-N. More frequently accessed data may be stored in the near region of a cache, with explicit or compiler generated hints being used to indicate the region that should be used to cache data upon initial insertion of the data into the cache. Hints can also be used to determine how readily data should be migrated between regions.

(428) FIG. **36** illustrates a package assembly for a parallel processor including a multi-region L3 cache. The package assembly of the compute system **3600** can include a parallel processor **3620** in which the various components of the parallel processor SOC are distributed across multiple chiplets. Each chiplet can be a distinct IP core that is independently designed and configured to communicate with other chiplets via one or more common interfaces. The chiplets may each be distinct and at least partially packaged integrated circuits that enables the parallel processor to be customized during assembly instead of during the design phase. The chiplets include but are not limited to compute chiplets **3605**, a media chiplet **3604**, and memory chiplets including near cache chiplets **3606**. The compute system **3600** may also include DRAM memory chiplets (not shown). Each chiplet can be separately manufactured using different process technologies. For example, compute chiplets **3605** may be manufactured using the smallest or most advanced process technology available at the time of fabrication, while cache chiplets **3606** or other chiplets (e.g., I/O, networking, etc.) may be manufactured using a larger or less advanced process technologies.

(429) The various chiplets can be bonded to a base die **3610** and configured to communicate with each other and logic within the base die **3610** via an interconnect layer **3612**. In one embodiment, the base die **3610** can include global logic **3601**, which can include scheduler **3611** and power management **3621** logic units, an interface **3602**, a dispatch unit **3603** that are interconnected via an interconnect fabric module **3608**. The interconnect fabric module **3608** may also enable communication between the global logic **3601** and the compute chiplets **3605**, media chiplet **3604**, and near cache chiplets **3606**. The interconnect fabric module **3608** can be an inter-chiplet fabric that is integrated into the base die **3610**. Logic chiplets can use the interconnect fabric module **3608** to relay messages between the various chiplets.

(430) The near cache chiplets **3606** and embedded far cache **3609** can operate in concert to provide an L3 cache to the media chiplet **3604** and the compute chiplets **3605**. The interconnect fabric module **3608** may be used to also enable communication between an embedded far cache **3609** and the near cache chiplets **3606**. Alternatively, the near cache chiplets **3606** may be directly coupled with the embedded far cache **3609** via a silicon bridge or interconnect structure. The near/far L3 cache can be used cache data read from and transmitted to system memory of a host and/or any DRAM chiplets that may be included in the parallel processor **3620**.

(431) In one embodiment the global logic **3601** is a microcontroller that can execute firmware to

perform scheduler **3611** and power management **3621** functionality for the parallel processor **3620**. The microcontroller that executes the global logic can be tailored for the target use case of the parallel processor **3620**. The scheduler **3611** can perform global scheduling operations for the parallel processor **3620**. The power management **3621** functionality can be used to enable or disable individual chiplets within the parallel processor when those chiplets are not in use.
(432) The various chiplets of the parallel processor **3620** can be designed to perform specific functionality that, in existing designs, would be integrated into a single die. A set of compute chiplets **3605** can include clusters of compute units (e.g., execution units, streaming multiprocessors, etc.) that include programmable logic to execute compute or graphics shader instructions. A media chiplet **3604** can include hardware logic to accelerate media encode and decode operations. Memory chiplets may also be included that acts as local memory for the parallel processor **3620**.
(433) FIG. **37** illustrate a method **3700** of managing a cache having multiple cache regions. Method **3700** can be performed by cache management logic in a cache controller **3406** as in FIG. **34**. The cache management logic can make use of one or more counter arrays **3417** and/or cache replacement/migration logic **3427**.
(434) Method **3700** includes for the cache management logic to receive a request associated with an access to data at a memory address, where the memory address is associated with a cacheable memory allocation (**3702**). The cache management logic can then check for a cache hit for the data within a region of a multi-region cache (**3703**). The cache management logic can include metadata to track which tags are stored in which regions (e.g., near, far) of the multi-region cache or can check each region in parallel.
(435) Upon a cache hit (**3704**, NO), the cache management logic can read data from the region associated with the cache hit (**3707**). Upon a cache miss (**3704**, YES), the cache management logic can read data from memory or a higher level cache (**3706**). For example, if the multi-region cache is an L3 cache, the data can be read from an LLC cache. Otherwise the data can be read from memory.
(436) The cache management logic can then determine if the region to which the data is to be stored is full (**3708**). A default region for newly added data may be used (e.g., a near region). The region to which the data is stored may also be determined, for example, based on shader kernel hints. The shader kernel hints can be compiler generated hints or explicit hints provided by a developer of the shader kernel. The shader kernel hints can indicate that the data in question will be frequently accessed or infrequently accessed. Data that is hinted to be infrequently accessed may be stored to the far region, while data that is hinted to be frequently accessed may be stored to the near region.
(437) If the region to which the data is to be stored is not full (**3708**, NO), the cache management logic can store the data into a region of the multi-region cache (**3712**), where the region to which the data to be stored is may be determined based on a hint or a default region may be selected. If the region to which the data is to be stored is full, the cache management logic can evict data from the region according to a cache replacement policy (**3710**) before storing the data to the region of the cache.
(438) After the cache management logic reads data from the region associated with the cache hit (**3707**) or evicts data from the region according to a cache replacement policy (**3710**), the cache management logic may migrate data between regions according to cache migration policy (**3709**). For example, frequently accessed data in the far region may be exchanged with less frequently accessed data in the near region. Migration may also occur as a result of the cache replacement policy, for example, if the SLRU replacement policy is in use.
(439) Using the techniques above, one skilled in the art may implement, for example, a data processing system comprising a general-purpose graphics processor including a single level of cache memory including a near region and a far region. The far region has a lower capacity and

lower latency relative to the far region. The data processing system additionally comprises a cache controller coupled with the cache memory. The cache controller includes region selection logic to select from one of the near region and the far region to store a cached unit of data. In one embodiment the region selection logic is configured to select from one of the near region and the far region to store the cached unit of data based on usage metrics associated with the cached unit of data. The region selection logic may also select from one of the near region and the far region to store the cached unit of data based on a hint associated with the unit of data. The hint associated with the unit of data may be generated during compilation of a shader kernel to be executed by the general-purpose processor. The hint may be generated in response to an instruction within the shader kernel. The instruction may indicate a usage frequency for the cached unit of data.

(440) Additional Exemplary Data Processing Systems and Computing Devices

(441) FIG. **38** is a block diagram of a data processing system **3800**, according to an embodiment. The data processing system **3800** is a heterogeneous processing system having a processor **3802**, unified memory **3810**, and a GPGPU **3820** including machine learning acceleration logic. The processor **3802** and the GPGPU **3820** can be any of the processors and GPGPU/parallel processors as described herein. The processor **3802** can execute instructions for a compiler **3815** stored in system memory **3812**. The compiler **3815** executes on the processor **3802** to compile source code **3814**A into compiled code **3814**B. The compiled code **3814**B can include instructions that may be executed by the processor **3802** and/or instructions that may be executed by the GPGPU **3820**. During compilation, the compiler **3815** can perform operations to insert metadata, including hints as to the level of data parallelism present in the compiled code **3814**B and/or hints regarding the data locality associated with threads to be dispatched based on the compiled code **3814**B. The metadata can also include hints related to a near or far cache region to which the thread data would preferably be stored. The compiler **3815** can include the information necessary to perform such operations or the operations can be performed with the assistance of a runtime library **3816**. The runtime library **3816** can also assist the compiler **3815** in the compilation of the source code **3814**A and can also include instructions that are linked at runtime with the compiled code **3814**B to facilitate execution of the compiled instructions on the GPGPU **3820**.

(442) The unified memory **3810** represents a unified address space that may be accessed by the processor **3802** and the GPGPU **3820**. The unified memory can include system memory **3812** as well as GPGPU memory **3818**. The GPGPU memory **3818** is memory within an address pace of the GPGPU **3820** and can include some or all of system memory **3812**. In one embodiment the GPGPU memory **3818** can also include at least a portion of any memory dedicated for use exclusively by the GPGPU **3820**. In one embodiment, compiled code **3814**B stored in system memory **3812** can be mapped into GPGPU memory **3818** for access by the GPGPU **3820**.

(443) The GPGPU **3820** includes multiple compute blocks **3824**A-**3824**N, which can include one or more of a variety of processing resources described herein. The processing resources can be or include a variety of different computational resources such as, for example, execution units, compute units, streaming multiprocessors, graphics multiprocessors, or multi-core groups. In one embodiment the GPGPU **3820** additionally includes a tensor accelerator **3823**, which can include one or more special function compute units that are designed to accelerate a subset of matrix operations (e.g., dot product, etc.). The tensor accelerator **3823** may also be referred to as a tensor accelerator or tensor core. In one embodiment, logic components within the tensor accelerator **3823** may be distributed across the processing resources of the multiple compute blocks **3824**A-**3824**N.

(444) The GPGPU **3820** can also include a set of resources that can be shared by the compute blocks **3824**A-**3824**N and the tensor accelerator **3823**, including but not limited to a set of registers **3825**, a power and performance module **3826**, and a cache **3827**. In one embodiment the registers **3825** include directly and indirectly accessible registers, where the indirectly accessible registers are optimized for use by the tensor accelerator **3823**. The power and performance module **3826** can be configured to adjust power delivery and clock frequencies for the compute blocks **3824**A-**3824**N

to power gate idle components within the compute blocks **3824**A-**3824**N. In various embodiments the cache **3827** can include an instruction cache and/or a lower level data cache.

(445) The GPGPU **3820** can additionally include an L3 data cache **3830**, which can be used to cache data accessed from the unified memory **3810** by the tensor accelerator **3823** and/or the compute elements within the compute blocks **3824**A-**3824**N. In one embodiment the L3 data cache **3830** includes shared local memory **3832** that can be shared by the compute elements within the compute blocks **3824**A-**3824**N and the tensor accelerator **3823**.

(446) In one embodiment the GPGPU **3820** includes instruction handling logic, such as a fetch and decode unit **3821** and a scheduler controller **3822**. The fetch and decode unit **3821** includes a fetch unit and decode unit to fetch and decode instructions for execution by one or more of the compute blocks **3824**A-**3824**N or the tensor accelerator **3823**. The instructions can be scheduled to the appropriate functional unit within the compute block **3824**A-**3824**N or the tensor accelerator via the scheduler controller **3822**. In one embodiment the scheduler controller **3822** is an ASIC configurable to perform advanced scheduling operations. In one embodiment the scheduler controller **3822** is a micro-controller or a low energy-per-instruction processing core capable of executing scheduler instructions loaded from a firmware module.

(447) In one embodiment some functions to be performed by the compute blocks **3824**A-**3824**N can be directly scheduled to or offloaded to the tensor accelerator **3823**. In various embodiments the tensor accelerator **3823** includes processing element logic configured to efficiently perform matrix compute operations, such as multiply and add operations and dot product operations used by 3D graphics or compute shader programs. In one embodiment the tensor accelerator **3823** can be configured to accelerate operations used by machine learning frameworks. In one embodiment the tensor accelerator **3823** is an application specific integrated circuit explicitly configured to perform a specific set of parallel matrix multiplication and/or addition operations. In one embodiment the tensor accelerator **3823** is a field programmable gate array (FPGA) that provides fixed function logic that can updated between workloads. The set of matrix operations that can be performed by the tensor accelerator **3823** may be limited relative to the operations that can be performed by the compute block **3824**A-**3824**N. However, the tensor accelerator **3823** can perform those the operations at a significantly higher throughput relative to the compute block **3824**A-**3824**N.

(448) FIG. **39** is a block diagram of a computing device **3900** including a graphics processor **3904**, according to an embodiment. The computing device **3900** can be a computing device that includes functionality of each of the embodiments described above. The computing device **3900** may be or be included within a communication device such as a set-top box (e.g., Internet-based cable television set-top boxes, etc.), global positioning system (GPS)-based devices, etc. The computing device **3900** may also be or be included within mobile computing devices such as cellular phones, smartphones, personal digital assistants (PDAs), tablet computers, laptop computers, e-readers, smart televisions, television platforms, wearable devices (e.g., glasses, watches, bracelets, smartcards, jewelry, clothing items, etc.), media players, etc. For example, in one embodiment, the computing device **3900** includes a mobile computing device employing an integrated circuit (“IC”), such as system on a chip (“SoC” or “SOC”), integrating various hardware and/or software components of computing device **3900** on a single chip.

(449) The computing device **3900** includes a graphics processor **3904**. The graphics processor **3904** represents any graphics processor described herein. The graphics processor includes one or more graphics engine(s), graphics processor cores, and other graphics execution resources as described herein. Such graphics execution resources can be presented in the forms including but not limited to execution units, shader engines, fragment processors, vertex processors, streaming multiprocessors, graphics processor clusters, or any collection of computing resources suitable for the processing of graphics resources or image resources, or performing general purpose computational operations in a heterogeneous processor.

(450) In one embodiment, the graphics processor **3904** includes a cache **3914**, which can be a

single cache or divided into multiple segments of cache memory, including but not limited to any number of L1, L2, L3, or L4 caches, render caches, depth caches, sampler caches, and/or shader unit caches. The cache **3914** may have a near and far region as described herein. The cache **3914** may also include dynamic hash logic that supports dynamic reconfiguration of a memory bank hash algorithm. In some embodiments, the graphics processor **3904** includes a GPGPU engine **3944** that includes shared local memory (SLM **3934**), as well as a register file **3924**, including includes registers for use by the GPGPU engine **3944**. The register file **3924** can include general-purpose registers, architectural registers, configuration registers, and other types of registers. A general-purpose register file (GRF) and/or architectural register file (ARF) can also reside within processing resources within one or more blocks of compute units (e.g., compute **3950**, compute **3955**) within the GPGPU engine **3944**. A shared fabric **3942** may also be present that enables rapid communication between the various components of the GPGPU engine **3944**.

(451) As illustrated, in one embodiment, and in addition to the graphics processor **3904**, the computing device **3900** may further include any number and type of hardware components and/or software components, including, but not limited to an application processor **3906**, memory **3908**, and input/output (I/O) sources **3910**. The application processor **3906** can interact with a hardware graphics pipeline to share graphics pipeline functionality. Processed data is stored in a buffer in the hardware graphics pipeline and state information is stored in memory **3908**. The resulting data can be transferred to a display controller for output via a display device, such as the display device **1618** of FIG. **16**A. The display device may be of various types, such as Cathode Ray Tube (CRT), Thin Film Transistor (TFT), Liquid Crystal Display (LCD), Organic Light Emitting Diode (OLED) array, etc., and may be configured to display information to a user via a graphical user interface.

(452) The application processor **3906** can include one or processors and may be the central processing unit (CPU) that is used at least in part to execute an operating system (OS) **3902** for the computing device **3900**. For example, the application processor **3906** may be CPU **3002** as in FIG. **30**. The OS **3902** can serve as an interface between hardware and/or physical resources of the computing device **3900** and one or more users. The OS **3902** can include driver logic for various hardware devices in the computing device **3900**, including graphics driver logic **3922**, such as the user mode graphics driver **2326** and/or kernel mode graphics driver **2329** of FIG. **23**.

(453) It is contemplated that in some embodiments the graphics processor **3904** may exist as part of the application processor **3906** (such as part of a physical CPU package) in which case, at least a portion of the memory **3908** may be shared by the application processor **3906** and graphics processor **3904**, although at least a portion of the memory **3908** may be exclusive to the graphics processor **3904**, or the graphics processor **3904** may have a separate store of memory. The memory **3908** may comprise a pre-allocated region of a buffer (e.g., framebuffer); however, it should be understood by one of ordinary skill in the art that the embodiments are not so limited, and that any memory accessible to the lower graphics pipeline may be used. The memory **3908** may include various forms of random-access memory (RAM) (e.g., SDRAM, SRAM, etc.) comprising an application that makes use of the graphics processor **3904** to render a desktop or 3D graphics scene. A memory controller hub, such as, for example memory hub **105** of FIG. **1**, may access data in the memory **3908** and forward it to graphics processor **3904** for graphics pipeline processing. The memory **3908** may be made available to other components within the computing device **3900**. For example, any data (e.g., input graphics data) received from various I/O sources **3910** of the computing device **3900** can be temporarily queued into memory **3908** prior to their being operated upon by one or more processor(s) (e.g., application processor **3906**) in the implementation of a software program or application. Similarly, data that a software program determines should be sent from the computing device **3900** to an outside entity through one of the computing system interfaces, or stored into an internal storage element, is often temporarily queued in memory **3908** prior to its being transmitted or stored.

(454) The I/O sources can include devices such as touchscreens, touch panels, touch pads, virtual

or regular keyboards, virtual or regular mice, ports, connectors, network devices, or the like, and can attach via an I/O hub **107** as in FIG. **1**, Input/output (I/O) circuitry **363** as in FIG. **3**A, a platform controller hub **1430** as in FIG. **14**, or the like. Additionally, the I/O sources **3910** may include one or more I/O devices that are implemented for transferring data to and/or from the computing device **3900** (e.g., a networking adapter); or, for a large-scale non-volatile storage within the computing device **3900** (e.g., hard disk drive). User input devices, including alphanumeric and other keys, may be used to communicate information and command selections to graphics processor **3904**. Another type of user input device is cursor control, such as a mouse, a trackball, a touchscreen, a touchpad, or cursor direction keys to communicate direction information and command selections to GPU and to control cursor movement on the display device. Camera and microphone arrays of the computing device **3900** may be employed to observe gestures, record audio and video and to receive and transmit visual and audio commands.
(455) I/O sources **3910** configured as network interfaces can provide access to a network, such as a LAN, a wide area network (WAN), a metropolitan area network (MAN), a personal area network (PAN), Bluetooth, a cloud network, a cellular or mobile network (e.g., 3.sup.rd Generation (3G), 4.sup.th Generation (4G), 5.sup.th Generation (5G), etc.), a satellite network, an intranet, the Internet, etc. Network interface(s) may include, for example, a wireless network interface having one or more antenna(e). Network interface(s) may also include, for example, a wired network interface to communicate with remote devices via network cable, which may be, for example, an Ethernet cable, a coaxial cable, a fiber optic cable, a serial cable, or a parallel cable.
(456) Network interface(s) may provide access to a LAN, for example, by conforming to IEEE 802.11 standards, and/or the wireless network interface may provide access to a personal area network, for example, by conforming to Bluetooth standards. Other wireless network interfaces and/or protocols, including previous and subsequent versions of the standards, may also be supported. In addition to, or instead of, communication via the wireless LAN standards, network interface(s) may provide wireless communication using, for example, Time Division, Multiple Access (TDMA) protocols, Global Systems for Mobile Communications (GSM) protocols, Code Division, Multiple Access (CDMA) protocols, and/or any other type of wireless communications protocols.
(457) It is to be appreciated that a lesser or more equipped system than the example described above may be preferred for certain implementations. Therefore, the configuration of the computing device **3900** may vary from implementation to implementation depending upon numerous factors, such as price constraints, performance requirements, technological improvements, or other circumstances. Examples include (without limitation) a mobile device, a personal digital assistant, a mobile computing device, a smartphone, a cellular telephone, a handset, a one-way pager, a two-way pager, a messaging device, a computer, a personal computer (PC), a desktop computer, a laptop computer, a notebook computer, a handheld computer, a tablet computer, a server, a server array or server farm, a web server, a network server, an Internet server, a work station, a mini-computer, a main frame computer, a supercomputer, a network appliance, a web appliance, a distributed computing system, multiprocessor systems, processor-based systems, consumer electronics, programmable consumer electronics, television, digital television, set top box, wireless access point, base station, subscriber station, mobile subscriber center, radio network controller, router, hub, gateway, bridge, switch, machine, or combinations thereof.
(458) Embodiments may be implemented as any one, or a combination of one or more microchips or integrated circuits interconnected using a parent-board, hardwired logic, software stored by a memory device and executed by a microprocessor, firmware, an application specific integrated circuit (ASIC), and/or a field programmable gate array (FPGA). The term “logic” may include, by way of example, software or hardware and/or combinations of software and hardware.
(459) Embodiments may be provided, for example, as a computer program product which may include one or more machine-readable media having stored thereon machine-executable

instructions that, when executed by one or more machines such as a computer, network of computers, or other electronic devices, may result in the one or more machines carrying out operations in accordance with embodiments described herein. A machine-readable medium may include, but is not limited to, floppy diskettes, optical disks, CD-ROMs (Compact Disc-Read Only Memories), and magneto-optical disks, ROMs, RAMs, EPROMs (Erasable Programmable Read Only Memories), EEPROMs (Electrically Erasable Programmable Read Only Memories), magnetic or optical cards, flash memory, or other type of non-transitory machine-readable media suitable for storing machine-executable instructions.

(460) Moreover, embodiments may be downloaded as a computer program product, wherein the program may be transferred from a remote computer (e.g., a server) to a requesting computer (e.g., a client) by way of one or more data signals embodied in and/or modulated by a carrier wave or other propagation medium via a communication link (e.g., a modem and/or network connection).

(461) Reference herein to “one embodiment” or “an embodiment” means that a particular feature, structure, or characteristic described in conjunction with the embodiment can be included in at least one embodiment of the invention. The appearances of the phrase “in one embodiment” in various places in the specification do not necessarily all refer to the same embodiment. The processes depicted in the figures that follow can be performed by processing logic that comprises hardware (e.g. circuitry, dedicated logic, etc.), software (as instructions on a non-transitory machine-readable storage medium), or a combination of both hardware and software. Reference will be made in detail to various embodiments, examples of which are illustrated in the accompanying drawings. In the following detailed description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. However, it will be apparent to one of ordinary skill in the art that the present invention may be practiced without these specific details. In other instances, well-known methods, procedures, components, circuits, and networks have not been described in detail so as not to unnecessarily obscure aspects of the embodiments.

(462) It will also be understood that, although the terms first, second, etc. may be used herein to describe various elements, these elements should not be limited by these terms. These terms are only used to distinguish one element from another. For example, a first contact could be termed a second contact, and, similarly, a second contact could be termed a first contact, without departing from the scope of the present invention. The first contact and the second contact are both contacts, but they are not the same contact.

(463) The terminology used herein is for the purpose of describing particular embodiments only and is not intended to be limiting as to all embodiments. As used in the description of the invention and the appended claims, the singular forms “a”, “an” and “the” are intended to include the plural forms as well, unless the context clearly indicates otherwise. It will also be understood that the term “and/or” as used herein refers to and encompasses any and all possible combinations of one or more of the associated listed items. It will be further understood that the terms “comprises” and/or “comprising,” when used in this specification, specify the presence of stated features, integers, steps, operations, elements, and/or components, but do not preclude the presence or addition of one or more other features, integers, steps, operations, elements, components, and/or groups thereof.

(464) As used herein, the term “if” may be construed to mean “when” or “upon” or “in response to determining” or “in response to detecting,” depending on the context. Similarly, the phrase “if it is determined” or “if [a stated condition or event] is detected” may be construed to mean “upon determining” or “in response to determining” or “upon detecting [the stated condition or event]” or “in response to detecting [the stated condition or event],” depending on the context.

(465) Embodiments described herein provide techniques to enable the dynamic reconfiguration of memory on a general-purpose graphics processing unit. One embodiment described herein enables dynamic reconfiguration of cache memory bank assignments based on hardware statistics. One embodiment enables for virtual memory address translation using mixed four kilobyte and 64 kilobyte pages within the same page table hierarchy and under the same page directory. One

embodiment provides for a graphics processor and associated heterogenous processing system having near and far regions of the same level of a cache hierarchy.
(466) One embodiment provides for a general-purpose graphics processor comprising a cache memory including multiple memory banks and a cache controller coupled with the cache memory. The cache controller includes a dynamic hash unit to select, based on a first hash function and an address associated with a memory access, one or more of the multiple memory banks to store data associated with the memory access. The general-purpose graphics processor also includes cache monitor hardware to monitor an access pattern of the multiple memory banks. The cache monitor hardware can request the dynamic hash unit to select a second hash function for use to select the one or more of the multiple memory banks. The cache monitor hardware can request the selection of the second hash function in response to detection of a bank access pattern disparity over a threshold.
(467) One embodiment provides for a method comprising, on a general-purpose graphics processor including a cache memory having multiple memory banks and a cache controller coupled with the cache memory, selecting, via the cache controller, a first set of one or more memory banks of the cache memory to access in response to a first memory access request to a first address. The first set of one or more memory banks is selected based on an address associated with the memory access request and a hash of the memory address generated via a first hash function. The method additionally includes monitoring an access pattern to the multiple memory banks of the cache memory and in response to detection of a bank access disparity, requesting the cache controller to change from the first hash function to a second hash function. The method additionally includes selecting, via the cache controller, a second set of one or more memory banks of the cache memory to access in response to a second memory access request to the first address.
(468) In one embodiment the method additionally comprises receiving, at the cache controller, the request to change from the first hash function to the second hash function, flushing the cache memory, and changing from the first hash function to the second hash function. Flushing the cache memory includes writing modified cache lines to an additional memory coupled with the general-purpose graphics processor and invalidating all cache lines in the cache memory.
(469) In one embodiment the method additionally comprises monitoring a thermal state of the multiple memory banks and in response to detecting that a temperature of one or more of the multiple memory banks exceeds a threshold, requesting the cache controller to change to a third hash function that is different from the second hash function.
(470) In one embodiment the method additionally comprises receiving, at the cache controller, a request to flush the cache memory, flushing the cache memory, and after flushing the cache memory, changing to the second hash function. The method additionally comprises to change, after flushing the cache memory in response to a flush request, to change to the third hash function.
(471) Additional embodiments provide the graphics processing logic to perform the above indicated method. Additional embodiments also include a data processing system including the above indicated graphics processing logic. The above described techniques may be integrated within or adapted to any of the graphics or parallel processor architectures described herein.
(472) Those skilled in the art will appreciate from the foregoing description that the broad techniques of the embodiments can be implemented in a variety of forms. Therefore, while the embodiments have been described in connection with particular examples thereof, the true scope of the embodiments should not be so limited since other modifications will become apparent to the skilled practitioner upon a study of the drawings, specification, and following claims.

Claims

1. A general-purpose graphics processor comprising: an interface to a host processor; a memory interface; a processing array including a plurality of graphics processing resources, the processing

array coupled with a memory via the memory interface; and memory management circuitry configured to: receive a request to map to a 4 kilobyte page of virtual memory associated with the host processor; create a mapping in a page table hierarchy for the 4 kilobyte page; receive a request to map to a 64 kilobyte page of virtual memory associated with processing array; and create a mapping in the page table hierarchy for the 64 kilobyte page, wherein the page table hierarchy is to concurrently store entries for the 4 kilobyte page and the 64 kilobyte page.

2. The general-purpose graphics processor of claim 1, wherein the memory management circuitry is configured to facilitate a unified virtual memory system that includes memory of the host processor and memory coupled with the processing array via the memory interface.

3. The general-purpose graphics processor of claim 2, wherein the page table hierarchy includes a page table that is shared between the processing array and the host processor.

4. The general-purpose graphics processor of claim 3, wherein the page table that is shared between the processing array and the host processor is configured to support a 49-bit address space for the processing array and a 48-bit address space for the host processor.

5. The general-purpose graphics processor of claim 1, wherein the page table hierarchy includes a page directory pointer table, a page directory table, a first page table configure to store page table entries for 64 kilobyte pages, and a second page table configured to store page table entries for 64 kilobyte pages and 4 kilobyte pages.

6. The general-purpose graphics processor of claim 5, comprising a translation lookaside buffer to cache entries within the page table hierarchy and a page table walker to walk the page table hierarchy in response to a cache miss at the translation lookaside buffer.

7. The general-purpose graphics processor of claim 6, wherein the translation lookaside buffer is configurable to cache sixteen contiguous 4 kilobyte page table entries of the second page table as a 64 kilobyte page table entry.

8. A method comprising: interfacing with a host processor at a general-purpose graphics processor; receiving a request at memory management circuitry of the general-purpose graphics processor to map a 4 kilobyte page of virtual memory associated with the host processor; creating a mapping in a page table hierarchy for the 4 kilobyte page; receiving a request to map to a 64 kilobyte page of virtual memory associated with processing array; and creating a mapping in the page table hierarchy for the 64 kilobyte page, wherein the page table hierarchy is to concurrently store entries for the 4 kilobyte page and the 64 kilobyte page.

9. The method of claim 8, comprising facilitating, via the memory management circuitry of the general-purpose graphics processor, a unified virtual memory system that includes memory of the host processor and memory of the general-purpose graphics processor.

10. The method of claim 9, comprising sharing the page table hierarchy between the general-purpose graphics processor and the host processor.

11. The method of claim 10, comprising: receiving a request at the general-purpose graphics processor to translate a virtual address of a first memory allocation within a 4 kilobyte page; in response to a miss in a translation lookaside buffer of the general-purpose graphics processor for the virtual address for the first memory allocation, initiating a page walk into the page table hierarchy to determine a physical address of the first memory allocation; receiving a request at the general-purpose graphics processor to translate the virtual address of a second memory allocation within a 64 kilobyte page; and in response to a miss in a translation lookaside buffer of the general-purpose graphics processor for the virtual address for the second memory allocation, initiating a page walk into the page table hierarchy to determine a physical address of the second memory allocation, wherein the page table hierarchy stores a first page table entry for the 4 kilobyte page and a second page table entry for the 64 kilobyte page.

12. The method of claim 11, comprising: storing the first page table entry in a first page table of the page table hierarchy, the first page table configured to store page table entries for 4 kilobyte pages; and storing the second page table entry in a second page table of the page table hierarchy, the

second page table configured to store page table entries for 4 kilobyte pages and 64 kilobyte pages.

13. The method of claim 11, comprising storing the first page table entry and the second page table entry in a page table configured to store page table entries for 4 kilobyte pages and 64 kilobyte pages.

14. The method of claim 13, comprising marking sixteen consecutive page table entries for 4 kilobyte pages as cacheable within a translation lookaside buffer as a single 64 kilobyte page.

15. A data processing system comprising: a host processor; and a general-purpose graphics processor coupled with the host processor via a host interface, the general-purpose graphics processor including: a memory device; a processing array including a plurality of graphics processing resources, the processing array coupled with a memory device via a memory interface; and memory management circuitry configured to: receive a request to map to a 4 kilobyte page of virtual memory associated with the host processor; create a mapping in a page table hierarchy for the 4 kilobyte page; receive a request to map to a 64 kilobyte page of virtual memory associated with processing array; and create a mapping in the page table hierarchy for the 64 kilobyte page, wherein the page table hierarchy is to concurrently store entries for the 4 kilobyte page and the 64 kilobyte page.

16. The data processing system of claim 15, wherein the memory management circuitry is configured to facilitate a unified virtual memory system that includes the memory device and memory of the host processor.

17. The data processing system of claim 16, wherein the page table hierarchy includes a page table that is shared between the processing array and the host processor.

18. The data processing system of claim 17, wherein the page table that is shared between the processing array and the host processor is configured to support a 49-bit address space for the processing array and a 48-bit address space for the host processor.

19. The data processing system of claim 15, wherein the page table hierarchy includes a page directory pointer table, a page directory table, a first page table configure to store page table entries for 64 kilobyte pages, and a second page table configured to store page table entries for 64 kilobyte pages and 4 kilobyte pages.

20. The data processing system of claim 19, comprising: a translation lookaside buffer to cache entries within the page table hierarchy; and a page table walker to walk the page table hierarchy in response to a cache miss at the translation lookaside buffer, wherein the translation lookaside buffer is configurable to cache sixteen contiguous 4 kilobyte page table entries of the second page table as a 64 kilobyte page table entry.