How General-Purpose can a GPU be?

Philip Machanick p.machanick@ru.ac.za
Department of Computer Science, Rhodes University, South Africa

If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?
Seymour Cray

The use of graphics processing units (GPUs) in general-purpose computation (GPGPU) is a growing field. GPU instruction sets, while implementing a graphics pipeline, draw from a range of single instruction multiple datastream (SIMD) architectures characteristic of the heyday of supercomputers. Yet only one of these SIMD instruction sets has been of application on a wide enough range of problems to survive the era when the full range of supercomputer design variants was being explored: vector instructions.

Supercomputers covered a range of exotic designs such as hypercubes and the Connection Machine (Fox, 1989). The latter is likely the source of the snide comment by Cray: it had thousands of relatively low-speed CPUs (Tucker & Robertson, 1988). Since Cray won, why are we not basing our ideas on his designs (Cray Inc., 2004), rather than those of the losers? The Top 500 supercomputer list is dominated by general-purpose CPUs, and nothing like the Connection Machine that headed the list in 1993 still exists[1].

In other words: given the lessons of the supercomputer era, why do GPGPU programmers have to grapple with the complexities of instruction sets that we *know* fit very few problems based on prior experience of highly data-parallel machines of the 1980s? The answer is that a GPU is a highly substitutable part: CPU designers are constrained by the code base out there, whereas a GPU only needs new graphics drivers to exploit a new, exotic instruction set. This means that GPU designers are not constrained from trying out new ideas, including instruction modes that fit the very narrow niche of implementing a graphics pipeline. Because GPUs are a huge market, these devices with exotic, highly parallel instruction sets offer an enticing opportunity for those seeking more speedup than is available in any other comparatively low-cost part.

How successful is GPGPU programming? Speedups reported in the GPGPU literature vary widely from less than 10 (Merrill & Grimshaw, 2010; Hughes & White, 2013) to over 100 (Krüger, Maitre, Jiménez, Baumes & Collet, 2010). Let us take this as our target: a speedup of over a conventional CPU of up to about 100 – while keeping in mind the need to implement a graphics pipeline.

Machanick, P. (2015). How General-Purpose can a GPU be? [Viewpoint]. *South African Computer Journal 57*, 113–117. http://dx.doi.org/10.18489/sacj.v0i57.347

[1] http://www.top500.org

An architecture with 100 cores that uses a relatively accessible programming model should in principle be competitive with all but the best cases for GPGPU, provided each core is reasonably fast. If we stick with a proven model, a shared-memory multiprocessor with a fast interconnect is amenable to a wide range of programming problems including highly parallel problems, multitasking workloads and moderately parallel problems (possibly using a subset of the CPUs). A reasonably large on-chip SRAM to mask the speed gap of going off chip will also be necessary to sustain most workloads (Machanick, Salverda & Pompe, 1998). If we add in vector instructions, the design can still be kept relatively simple, making it possible to scale to this number of cores within the budget of today's high-end GPUs (7-billion transistors surpassed in 2012 (NVIDIA, 2012; Chen, 2013)).

Communication is an issue with anything but a small number of processors. Over about 64 cores, uniform-latency interconnects become impractical; something closer to a traditional network with variable latency becomes a better design compromise (Sewell et al., 2012). Network-on-chip (NoC) (Hemani et al., 2000; Goossens, Dielissen & Radulescu, 2005; Pande, Grecu, Jones, Ivanov & Saleh, 2005; Ogras & Marculescu, 2013; Ginosar & Chatha, 2014) can scale to the required number of processors (Bjerregaard & Mahadevan, 2006).

Intel has explored part of the design space with the Larrabee architecture, which was based on multiple in-order multiple-issue Pentium cores with limited extra extensions to support graphics (Seiler et al., 2008). A design using a simpler RISC instruction set without the complexity of multiple issue would make it possible to implement more cores with the same transistor count. Intel abandoned the Larrabee strategy; they more recently have introduced the Xeon Phi multicore coprocessor, which features a specialist instruction set including vector modes to support high-performance computing (Heinecke et al., 2013). Unlike the Phi, the idea here is to implement a design that can also implement a graphics pipeline. The Phi is based on Intel Atom (a low-power variant of the x86 architecture) cores with a vector unit added to each.

Intel's latest Knights Landing version of the Phi features up to 72 cores, indicating that a design of the scale contemplated here is feasible (Gardner, 2014).

What does a GPU pipeline do? In its original form it was a static sequence of stages; recent designs are more programmable. The major stages are (Luebke & Humphreys, 2007):

- *input* – usually in the form of primitives, e.g., OpenGL, that provide vertices, which the pipeline assembles into triangles
- *model transformations* – produces a stream of triangles in a unified coordinate system
- *lighting* – the triangles are coloured based on the lighting of the scene; this stage requires vector computations
- *camera simulation* – the GPU projects the scene onto the film plane of a virtual camera, producing a stream of triangles in screen coordinates; vector computation is again needed here
- *rasterization* – triangles that overlap screen pixels are calculated, and this is a highly parallel stage since each pixel can be handled independently

- *texturing* – images called textures are added to the near-final pixel colouring; this is also a highly parallel step and has a very regular memory access pattern
- *hidden surfaces* – pixels obscured by others have to be discarded, using a depth buffer that records how close a pixel is to the viewer and hence whether it can overwrite another pixel in the same spot on the screen

The main research question to be answered is whether the proposed design can implement a competitive graphics pipeline, with roughly the same component count as a GPU. If the graphics pipeline can be implemented with modes of parallelism no more exotic than a large number of conventional cores possibly with vector units, GPGPU becomes truly general-purpose. The challenge is to implement the highly parallel stages of the graphics pipeline and the aspects that lend themselves to specialist memory without using features that are difficult to apply to general programming.

Further, if we can get this right, the new design can leverage the key advantage of a GPU: the fact that it is a highly substitutable part in a large market. By contrast with the Xeon Phi (which only targets the compute-intensive market), provided the design can gain a significant foothold in the graphics market, it will achieve economies of scale that will make it viable for smaller niches like supercomputers.

Narrowing the research question to testing viability of a graphics pipeline with such a design avoids some of the harder questions, such as implementing a memory hierarchy for general workloads. The value in starting with the graphics pipeline is simplification of simulation studies – rather than simulating a workload with millions or billions of instruction executions spread over of the order of 100 cores, the simulation study only needs to show that the graphics pipeline can be implemented with typical operations within required latency targets.

Simulation is possible with existing research simulators, such as Gem5, which includes a capability of simulating a network with accurate timing (Binkert et al., 2011).

Finally, to show viability, enough of the logic needs to be designed to show that the proposed design is competitive in terms of component count with a comparable GPU. Part of this can be done by estimating CPU component count from previous designs of similar complexity, such as early RISC designs. For example, the MIPS R4000 is a single-issue design with a full 64-bit instruction set, and it required only 1.2-million transistors for the CPU and first-level cache (Mirapuri, Woodacre & Vasseghi, 1992).

ACKNOWLEDGMENTS

I would like to thank anonymous SAICSIT 2015 reviewers who rejected a version of this paper but made insightful comments on how to take the project forward, and the *SACJ* reviewers who provided further helpful comments. This work was undertaken in the Distributed Multimedia CoE at Rhodes University, with financial support from Telkom SA, Tellabs, Genband, Easttel, Bright Ideas 39, THRIP and NRF SA (TP13070820716). The author acknowledges that opinions, findings and conclusions or recommendations expressed here are those of the author and that none of the above mentioned sponsors accept liability whatsoever in this regard.

References

Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., . . . Wood, D. A. (2011, August). The Gem5 simulator. *SIGARCH Comput. Archit. News*, *39*(2), 1–7. http://dx.doi.org/10.1145/2024716.2024718

Bjerregaard, T. & Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computing Surveys*, *38*(1), 1.

Chen, J. Y. (2013). Transformation of VLSI technologies, systems and applications: The rise of foundry and its ecosystem. In *Int. Symp. on VLSI Technology, Systems, and Applications (VLSI-TSA)* (pp. 1–2). IEEE.

Cray Inc. (2004). *Optimizing applications on the Cray X1 system* (tech. rep. No. S-2315-52). Cray Inc. http://docs.cray.com/books/S-2315-52/html-S-2315-52/index.html.

Fox, G. C. (1989). Parallel computing comes of age: Supercomputer level parallel computations at Caltech. *Concurrency: Practice and Experience*, *1*(1), 63–103.

Gardner, E. (2014, November). What public disclosures has Intel made about Knights Landing? https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing.

Ginosar, R. & Chatha, K. S. (2014). Guest Editors' Introduction—Special Issue on Network-on-Chip. *IEEE Trans. on Computers*, *63*(3), 527–528.

Goossens, K., Dielissen, J. & Radulescu, A. (2005). Æthereal network on chip: concepts, architectures, and implementations. *IEEE Design & Test of Computers*, *22*(5), 414–421.

Heinecke, A., Vaidyanathan, K., Smelyanskiy, M., Kobotov, A., Dubtsov, R., Henry, G., . . . Dubey, P. (2013). Design and implementation of the Linpack benchmark for single and multi-node systems based on Intel Xeon Phi coprocessor. In *IEEE 27th Int. Symp. on Parallel & Distributed Processing (IPDPS)* (pp. 126–137).

Hemani, A., Jantsch, A., Kumar, S., Postula, A., Oberg, J., Millberg, M. & Lindqvist, D. (2000). Network on chip: An architecture for billion transistor era. In *Proc. IEEE NorChip Conf.* (Vol. 31).

Hughes, J. D. & White, J. T. (2013). Use of general purpose graphics processing units with MODFLOW. *Groundwater*, *51*(6), 833–846.

Krüger, F., Maitre, O., Jiménez, S., Baumes, L. & Collet, P. (2010). Speedups between × 70 and × 120 for a generic local search (memetic) algorithm on a single GPGPU chip. In *Applications of Evolutionary Computation* (pp. 501–511). Springer.

Luebke, D. & Humphreys, G. (2007). How GPUs work. *Computer*, *40*(2), 96–100.

Machanick, P., Salverda, P. & Pompe, L. (1998). Hardware-software trade-offs in a Direct Rambus implementation of the RAMpage memory hierarchy. In *Proc. 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)* (pp. 105–114). San Jose, California, USA. http://dx.doi.org/10.1145/291069.291032

Merrill, D. G. & Grimshaw, A. S. (2010). Revisiting sorting for GPGPU stream architectures. In *Proc. 19th Int. Conf. on Parallel Architectures and Compilation Techniques* (pp. 545–546). ACM.

Mirapuri, S., Woodacre, M. & Vasseghi, N. (1992). The Mips R4000 processor. *IEEE Micro*, *12*(2), 10–22.

NVIDIA. (2012). *NVIDIA's next generation CUDA compute architecture: Kepler GK110*. NVIDIA. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

Ogras, U. Y. & Marculescu, R. (2013). *Modeling, analysis and optimization of network-on-chip communication architectures*. Springer Science & Business Media.

Pande, P. P., Grecu, C., Jones, M., Ivanov, A. & Saleh, R. (2005). Performance evaluation and design trade-offs for network-on-chip interconnect architectures. *IEEE Trans. on Computers*, *54*(8), 1025–1040.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., ... Cavin, R. et al. (2008). Larrabee: A many-core x86 architecture for visual computing. In *Proc. SIGGRAPH '08*. ACM.

Sewell, K., Dreslinski, R. G., Manville, T., Satpathy, S., Pinckney, N., Blake, G., ... Sylvester, D. et al. (2012). Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, *2*(2), 278–294.

Tucker, L. & Robertson, G. (1988, August). Architecture and applications of the Connection Machine. *Computer*, *21*(8), 26–38. http://dx.doi.org/10.1109/2.74