Efficient Parallelism

Analysis of a High Performance per Watt Multiprocessor MIMD Architecture

COIS 4350H
Simon Willshire (0491272)

It is no news that parallelism is the solution to the problem of maintaining performance improvements in the high performance computing industry, and the consumer market in general. Whenever parallelism is implemented, communication overhead is the new problem that was previously memory and frequency wall that we have overcome by parallelism. However, the implementation of parallel solutions still has been fraught with cooling and of course communication overhead issues. One attempt at solving this is to use a very low power, reduced instruction set and very high core count. This method provides unprecedented performance per watt; this attempt was made by Adapteva with their Epiphany architecture. This paper will discuss mainly the design decisions that the Epiphany architecture utilizes, however we will also cover how the hardware market for small vendors has become increasingly difficult. How was it that Adapteva was able to bring their chips to market?

Adapteva was founded in 2008 by Andreas Olofsson. He has had years of knowledge in the industry, but wanted to make a difference in very low power solutions that produce high GFLOPS/watt architectures. Not only did he want this architecture to be highly efficient, but also very scalable for the high performance computing industry. He took a year in research and development, and came up with the Epiphany architecture. This architecture provides communication between a maximum of 4096 addressable RISC cores, and achieves his original efficiency goals. The first implementation of the architecture was created in 2011 named Parallela, it had 16 cores fabricated in 65nm and achieved 32GFLOPS. Three months later, a 64 core version using the same architecture was built in a downsized 28nm – and was the most energy efficient floating point processors in the world. However, this innovation was very difficult to bring to market, hardware start-up companies typically estimated 25-100 million USD to get off the ground. Adapteva used aggressive iterative design simplifications, modular chip design, horizontal engineering team integration, multi-project wafers in product design, and lastly utilized and built on common open source software to cut costs. These methods allowed them to bring 4 generations of the Epiphany chip to market with only 2 million in development costs. This was achieved through the use of a crowd funding website called Kickstarter – which brought them to their 1 million USD goal over the period of a month, and made the project possible.

Hardware Architecture

Overview Design Decisions

- Ease of use over complexity & performance
- Conforms to native **single precision** IEEE floating point instructions and format
- Conforms to ANSI-C99 byte addressing and programmability
- Provides registers for loop unrolling and variable reuse (64 entry capacity)
- 64bit Load/Store transfer from registers to memory: increasing memory bound math algorithms
- Provides additional systems for ease of use: Interrupt controller/debug unit and hardware timers (improve programmer implementation experience).

Overview Rejected Features

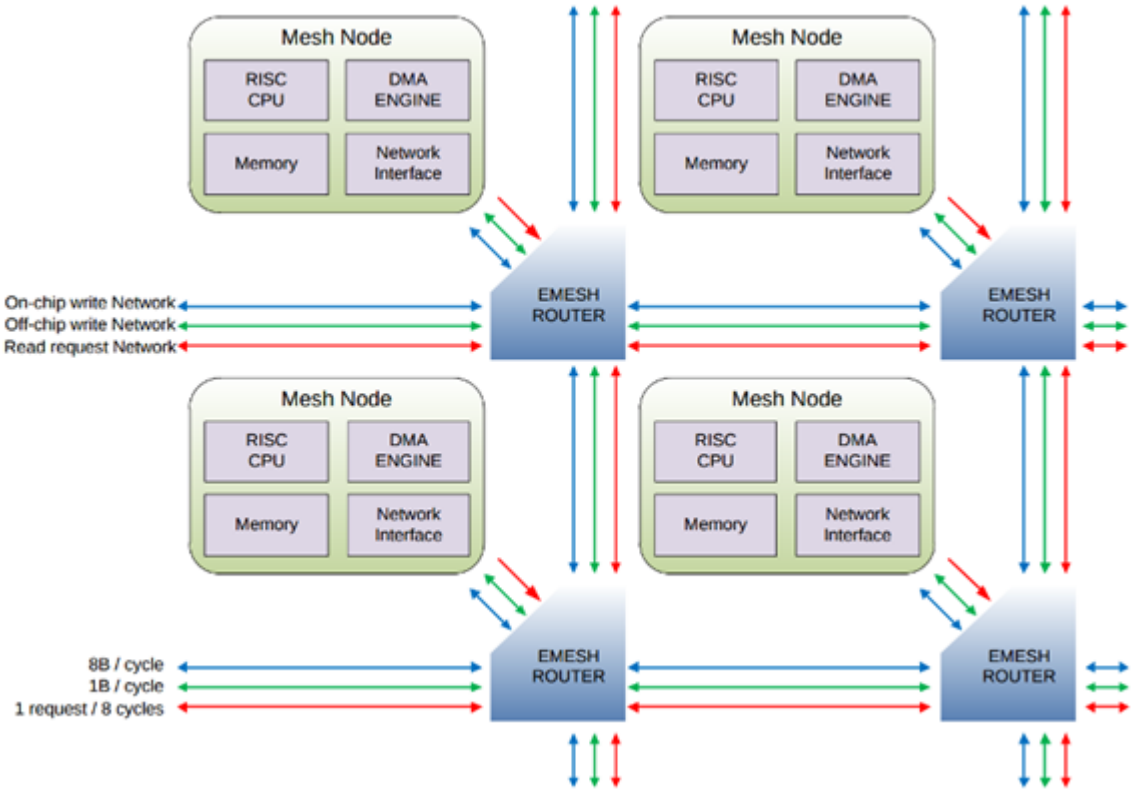
- Out of order scheduling (additional hardware caching: too much complexity/power consumption and use of additional off-chip memory access).
- Following instructions: NOT/MASK/ROTATE/ADD/CARRY Bit manipulation in specific instructions (Further reduction in complexity for ISA)
- Orthogonal ISA support - Hardware design complexity (Previous experience in TigerShark design influenced this decision): essentially using a single atomic memory allocation instead of segmenting different types, i.e. Unsigned/signed 8/16/32bit types, decided on a single base memory allocations.
- Simplified branch target buffer
    - Utilizes "branch never taken" with constant branch penalty when taken (reduced complexity, possibly reduced performance depending on implementation).
- Circuit switch based network: not flexible across all sets of applications.
- Stateless X/Y routing (enables scaling for large array sizes): "pushback signal propagation
    - Ensures no packet loss through input/output at consistent rate (equal input and output rate) at the cost of an extra stage of multiplex registers.
- Chosen to be benefit with single cycle hop and low stall penalties in processor pipeline.

Unique Design Decisions

- Communication mesh sends complete node address for every data transaction
    - Saves hardware overhead in place of design simplification.
- Completely flat memory hierarchy: single register file per node (32KB each)
    - Considered to be a cache only memory architecture (COMA).
- Mesh of RISC nodes using multiple instruction multiple data (MIMD) architecture.
- Maximum power use 2W (5V)

Communication

Adapteva uses what they have named the *eMesh* Network-On-Chip (NOC). The mesh connects all co-processor nodes in order for internode communication. The network packets include typical contents: source address, destination address and accompanying data. Each node can transfer up to 64 bytes of data through a mesh node every clock cycle, i.e. 64GB/s at typical operating frequency of 1 GHz per node.



eMesh Network-On-Chip (NOC)

As previously mentioned in the architecture overview, this architecture is shared-memory architecture – Each mesh node has without limitation, access to the rest of the memory in the other nodes in the mesh. However, to do so the architecture must transfer data through each of its nodes to from its origin node to its destination by nearest neighbour routing. What this means, is that each node must use one of its 4 connections in the mesh, and must pass data through the mesh if it is not the intended recipient.

The *eMesh* consists of three separate mesh structures for different types of transactions:

1. *cMesh*: Connects a mesh node to all of its neighbours with a maximum 8bytes/cycle in each operating direction (before stalling the pipeline). Used for network write transactions.
2. *rMesh*: Connects a mesh node to all of its neighbours with a maximum of a single read transaction every 8 clock cycles in each direction.
3. *xMesh*: Split into two networks, a north-south, and an east-west network. Used for write transactions that is not on the same chip (i.e., communication to host processor ARM A9). Total bandwidth of 8GB/sec in current models.

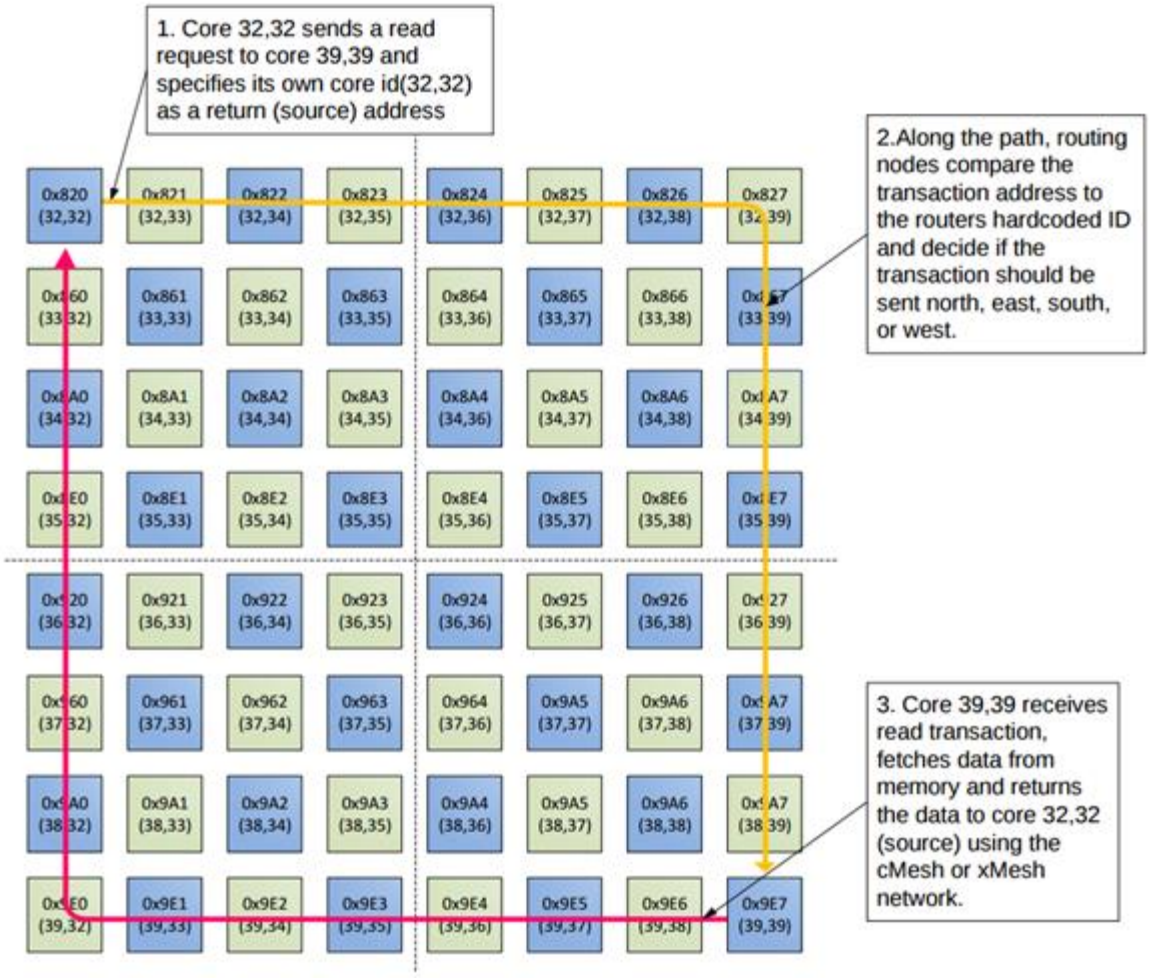
The *eMesh* was designed so that write transactions are 16x more efficient than read transactions, this ensures that memory consistency across all mesh nodes is faster and easier to achieve, and reduces the inter-node traffic on the mesh for read transactions. The separation of traffic through the use of the *xMesh* allows for decoupling of traffic from off-chip and on-chip communications. Perhaps one of the most important features this mesh network provides, is that it provides no deadlocks in all traffic conditions – This is made possible through the use of network separation, and the transaction movement along rows first, then columns. Lastly, as with any HPC implementation, scalability is a must in the design process. This mesh is capable of scaling to 4,096 processors in the current 32bit address map (shared memory system).

Inter-Node Communication

Due to the use of shared-memory architecture, each *eMesh* node can work on the same memory by simply setting a new address pointer to all other nodes that need an update. The hardware dictates what memory allocations belong to which nodes, the rest is seamless interaction. The *cMesh* network can operate at the same rate as the processor core, so there are no pipeline stalls, even while executing inter-node write transactions. With this method in mind, there is no need to use write based inter-node communication, it can all be done through shared memory interaction.

Routing Protocol

In order for proper node interaction in the *eMesh*, the propagation of data must first be defined. The node addresses contain a column tag as well as a row tag. These tags are compared as packets are passed along rows first, then columns. If a tag does not match, the packet is passed in the increasing/decreasing order until it reaches the addressed node. An example routing situation:



eMesh supports multicasting to many cores through the multicast routing mode. This is done through setting a configuration field in a configuration register in the mesh node you wish to multicast from. This overrides the typical routing protocol described above with one that propagates packets outwards.

Epiphany Processor Node

Each processor node in the network (*eCore*) consists of a RISC microprocessor, multi-bank local memory, and multicore-optimized direct memory access (DMA) engine which allows for timing the *eMesh* memory access with the CPU clock cycle. Each processor node contains local memory; this memory can be accessed in four ways simultaneously:

1. Instruction Fetch (8 byte instruction put into instruction decoder from program sequencer), can also fetch next instruction from current local memory, or another address location on the *eMesh*.
2. Load/Store: transfers data between register file, memory bank, and external memory.
3. DMA: similar to Load/Store, but manipulates mesh data, not register data
4. External: another node can access this nodes local memory at any time.

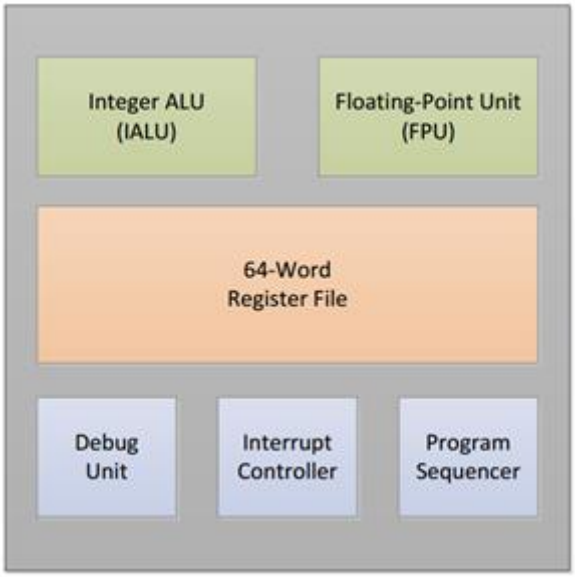
An event monitor has also been added in order to allow "debugging, optimization, load balancing, traffic balancing, timeout counting, watchdog timing, system time..." [1]. Lastly each node has a network interface which connects the node to the *eMesh*. It decodes "load and store instructions, program counter addresses and DMA-transaction addresses" [1].

The *eCore* CPU is a floating point RISC processor which is capable of a theoretical maximum of 2 GFLOPS. It includes a program sequencer, register file, integer (ALU), floating point unit (FPU), debug unit and interrupt controller - Typical CPU components to say the least.

Program Sequencer: Nothing spectacularly unique about this component, it assembles program instructions: loops, functions, jumps, interrupts, linear (feeds instructions from memory to pipeline without stalling).

Register File (64-Word): Perhaps a more interesting decision, each node provides temporary, power efficient memory access. Allows more temporary allocations; reducing overhead in access to main memory outside the mesh. Register access may simultaneously allow for the following in a single cycle:

- 3 floating point operands read, 1 result written by FPU
- 2 integer operands read, 1 result written by IALU
- Single 64bit double word written or read using load/store instruction (provides input output from register file).



Floating Point & Integer ALUs

Single cycle access to both units provided they are not accessing each other's data dependencies. Floating point data is single precision, and besides standard exception handling, adding, subtraction, multiplication, absolute and data conversions (between fixed and float data types), the FPU provides a "fused multiple-add, fused multiple-subtract" operands. These operands take three input operands and write the result to the register in a single cycle. These operands are intended for many multiply-accumulate operations [1]. We will discuss how useful these instructions are in the implementation section of the paper.

Other components are fairly typical, so we will glaze over them. Next we will discuss what really makes this architecture, the reduced and packed instruction set.

Instruction Set Architecture (ISA)

The instruction set used by Epiphany provides both 16bit and 32bit instructions; this means instructions can be packed. The method of packing is determined by which registers the instructions are stored, registers 0-7 are 16bit instructions, other registers are used for 32bit instructions.

Branch instructions and register jump instructions provide a method for conditional instruction execution. The architecture supports the comparison between results found on both arithmetic units on signed and unsigned data types – There are a total of 16 branching/condition codes (==, !=, >, >=, <, <=, ...)

Load/Store instructions consist of 3 different addressing modes: the memory address can be calculated by adding an offset for local variable access, or it may be incremented every index for array addressing, or it may be taken directly from the base register value, modified during instructions and reused afterward. The final mode is typically used for stacks/large data arrays.

These instructions are brief as they are very generic operations and have no unique qualities that separate them from a normal instruction architecture (with the exception of some memory management instructions due to architecture differences), but they are listed here for completions sake.

- Unsigned Integer Instructions; Add, Subtract, Shift Right/Left, Or, And, XOR, bit reverse operations.

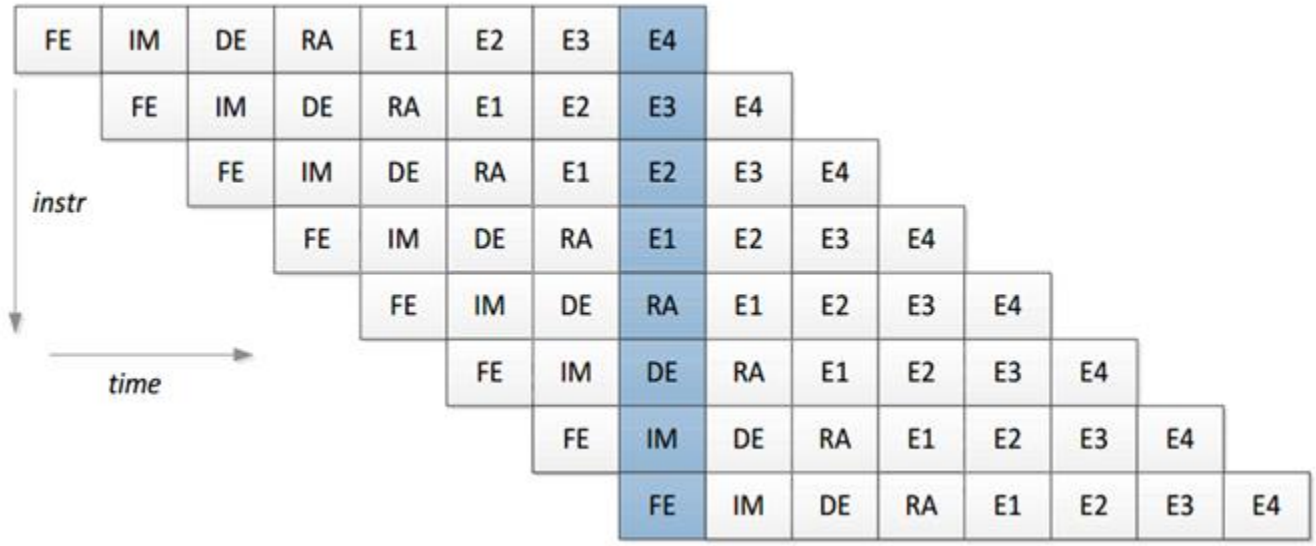
- Signed Integer Instructions; Add, Subtract, Multiply, Multiply-Add, Multiply-Subtract
- Floating Point Instructions; Add, subtract, multiply, multiply-add, multiply-subtract, conversions
- Register Move Instructions: Move immediate (on executing cores memory), and generic Move (from other core memory to local)

Instruction Pipeline Architecture

Due to the multi-size instruction length (16bit and 32 bit), the instruction pipeline includes a single branch based on this condition, otherwise it operates linearly by the following stages:

Note: Below pipeline stages have been taken *directly* from architecture reference document [1].

1. (FE) Fetch Address: address sent to instruction memory
2. (IM) Instruction Memory Access: Instruction returns from core memory
3. (DE) Decode: Instructions decoded
4. (RA) Register Access: Operands are read from register file for all instructions
5. (E1) Execution (16bit instructions completed after):
   - Load/Store address calculation (Register communication from mesh)
   - Register read for memory store operation.
   - Integer status flags written to register.
   - Branching & Jumps change program flow.
6. (E2) Execution: Data from load instruction written to register file.
7. (E3) Execution: Floating point result written to register file when truncation rounding mode set.
8. (E4) Execution: Floating point result written to register file when round to nearest even mode set.



Interlocked Pipeline

The classic RISC pipeline typically consists of 5 stages which is very similar to the above pipeline. The differences lie in the IM stage, where after the address has been fetched, the instruction is not immediately able to be accessed, it must be accessed in the second stage (As the memory could technically be stored on a completely different core). The second difference has already been mentioned above; where execute has been split into 4 clock cycles, the first of which can be prematurely exited depending on the instruction encoding and type.

An interlocking pipeline is common in RISC architectures; it restricts instructions before execution with two main defects. The first defect occurs due to structural referencing between two instructions, this may occur when

an instruction references an address, and so too does another instruction. This can be avoided by using branching, which provides a structure similar to order of operations in mathematics. The second hazard is quite common, since a typical clock cycle is much shorter that memory access, we can assume that we must wait for memory to become accessible after a previous instructions execution. The interlocking pipeline provides a solution to both of these problems by stalling the pipeline execution in order to accurately execute upcoming instructions in the correct order. Alternatively, instead of using an interlocking pipeline, you may reorder instructions without harming the integrity of program stack; this method is typically called operand forwarding. The Epiphany architecture uses an interlocked pipeline, which comes with benefits in terms of simplicity of design, however it does not provide as efficient results in terms of using all available cycles. However, the added hardware complexity and mainly the memory transfer rate on the Epiphany architecture make the interlocking solution (stalling pipeline) a good decision on Adapteva's part.

Dual Instruction Scheduling Protocol

Instruction Type	IALU	FPU/IALU2	LOAD/STORE	CONTROL
IALU		YES	NO	NO
FPU/IALU2	YES		YES	NO
LOAD/STORE	NO	YES		NO
CONTROL	NO	NO	NO	

The architecture allows two instructions to be scheduled at once in parallel on every clock cycle, provided the following protocol is followed: The instruction dispatch is done in order (they cannot read the same memory after a write or write to the same memory after a write – ensures data integrity). The architecture reference provides this handy chart for scheduling two operations at once:

Should instruction conditions create data-dependency issues; the processor will automatically stall stages of the pipeline in order to complete enough stages to continue processing. These stalls are typically mitigated using the customised C compiler (see Software section), however if the user chooses to write their own assembly instructions, it is important to remember these scheduling stages and how they impact the pipeline's performance, and therefore the systems performance in general.

Software Architecture

The importance to pair cooperative software and hardware technologies that do not conflict is essential to efficient system operation. Adapteva has opted to use very well-known open source technologies to accomplish this, with some modifications in order to effectively utilize their hardware. The open source technologies they chose allow them to first of all lessen the amount of learning curve for most users (such as the well-known GCC compiler as an example). It also allows them to benefit from support and fixes that the community of open source developers contribute to, and saves Adapteva operating costs in the long run. Adapteva is not a software company, and as such they should not have to focus on reinventing the wheel when it comes to software.

Programming Framework

Epiphany processor nodes may run independent programs via the use of a common loader. A multicore IDE (Eclipse recommended by Adapteva, again an open source multi-platform solution with wide spread support) handles the configuration between multicore projects to do this effectively. The SDK distribution is named

the *eSDK*, provided by Adapteva, which is installed on each Epiphany board you choose to work on. Eclipse IDE provides support for debugging directly on the Epiphany board from a host machine (x86/x64 processor required for this IDE). Debugging breakpoints can target specific operating cores, or on all cores [2].

Depending on where your preferences lie (command-line/system programmers), you may use a modified GCC compiler (coined *eGCC*), which supports from C++89 to C++11. For Epiphany architecture operation, Adapteva recommends the use of double world loop alignment (in order to properly flush the pipeline (see above pipeline information)).

The Epiphany assembler allows the user to compile assembly code directly, using the Epiphany ISA outline above, once this is compiled, it is sent to the linker in typical compilation fashion.
A unique feature added to the linking process, is the explicit code/data memory management options. A user may specify specific core row and core columns to execute on, as well as where to allocate stack/heap addresses during execution. These options can alternatively be written in your C code, which will be handled at a higher level before the linker gets a hold of your configuration. Memory management scenarios have been provided, where you can decide where to store user code [2].

Memory Management of User Code

User code can be stored preconfigured by the linker in three ways:

1. Legacy: All external SDRAM storage, usually just used for testing. User code, standard library and stack are stored and operate externally from the Epiphany memory.
2. Fast: All user code & data, and stack is stored internally, the standard library is stored externally. Typical use for fast critical functions, to ensure all operating code and data fits within local memory.
3. Internal: All code, static data, stack and standard library all stored on local memory in mesh. Fastest possible execution, however it is required that the user ensures everything can actually fit within internal memory...

These techniques give programmers very specific control over the operation of the program, and of course this adds complexity for superior performance in program operation. Following along with Epiphany's customised ISA (outlined above), Adapteva has provided a simulator for accurately testing the register address map for a single mesh core. It does not provide any pipeline simulation, but it does provide a multi-platform test environment for Epiphany simulation. The only method of debugging and profiling is through program traces – again catered towards systems programmers.

Similar to MPI and other message passing libraries, the *eSDK* is capable of segregating regions of cores into "workgroups" which can be defined at runtime. A workgroup can be a segment of one or multiple adjacently configured *eMeshes* (Multiple epiphany chips working in the same mesh through external communication).
In addition to workgroups, other runtime configurations can be set during runtime using preprocessor pragmas or assembly executing references. These functions can be included by using the *eSDK* header reference *e-lib.h* to retrieve hardware status during runtime. System registers, interrupt services, and other CPU features (as described above) can be accessed using the same inclusion reference [2].

Now that we have covered the basic features provided by the software and hardware components offered, we will demonstrate the effectiveness and efficiency offered by the Epiphany architecture.

Implementation & Benchmarks

Matrix Multiplication

The Epiphany IV (64 core model) was tested using a 512x512 matrix multiplication written in ANSI-C. This implementation was done in a linear 3 for loop fashion with some optimization on both x86 host side, and the Epiphany board.

Results []

- Epiphany IV (64 core): 219.8ms
- Host x86 Platform (AMD E-350) @ 1.6Ghz: ~4300ms

Face Detection & Image Processing

Using the popular OpenCV libraries with LBP face detection algorithm and comparing it to a reference x86 processor, with the following results [4]:

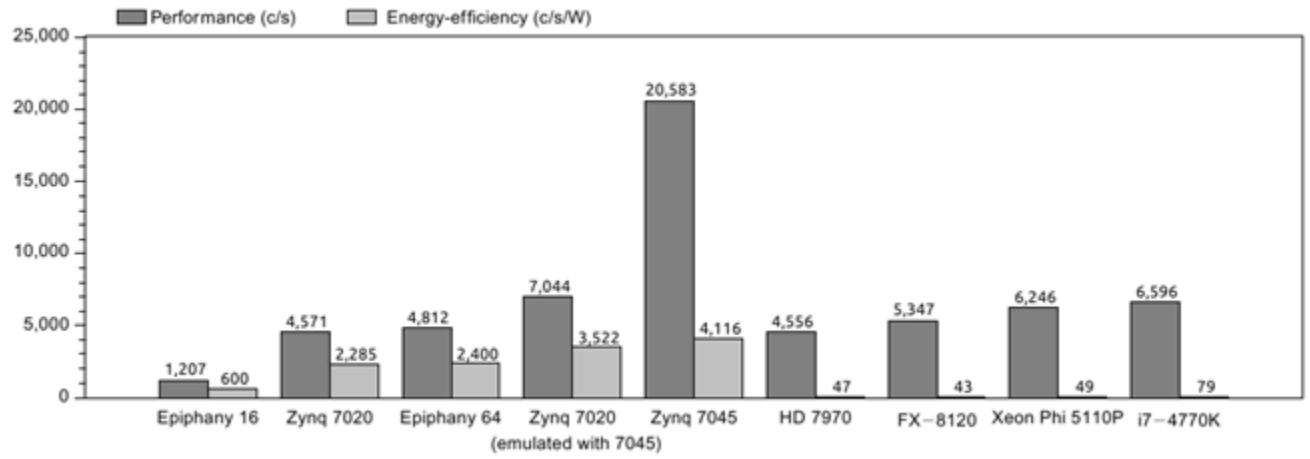
- x86 reference platform: 1340ms
- 1 Epiphany core: 16100ms
- 16 Epiphany cores: 1000ms

Cryptography

BCrypt cryptography implementation: (2 instances per core) written and optimized in Epiphany assembly.

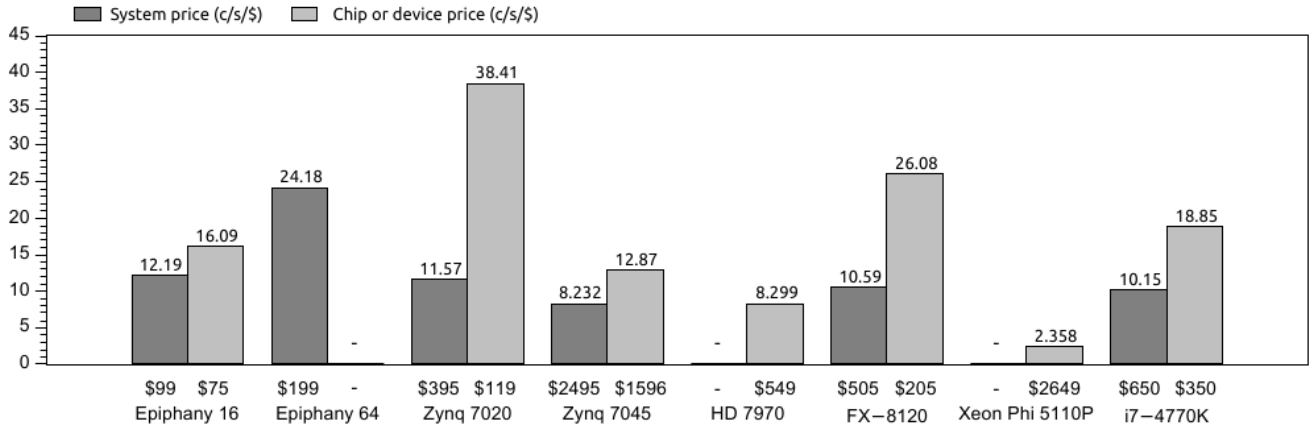
Results [3]

- 2400 cycles/sec/watt on the 64 core Epiphany board, 1000 cycles/sec/watt on 16 core models.
  - **Scaling is %99.7 efficient with 4x more processors (16 cores to the 64 core board).**
- FPGA implementation (a significant amount of work to write/implement and get right): using a modified (improved cooling & power supply stability) ZedBoard FPGA (model Zynq 7020): **2285** cycles/sec/watt.
- Desktop CPU: i7-4470K: **79** cycles/sec/watt (30x less efficient)
- Desktop GPU 1: HD 7990: **46** cycles/second/watt (52x less efficient)
- Desktop GPU 2: GTX TITAN: **6** cycle/second/watt (GPU's are not typically well suited for bcrypt implementations and due to large transistor count for die size, makes for poor power consumption in benchmarks).



BCrypt Implementation: Epiphany, FPGA, Desktop & Server hardware [3]

Clearly the FPGA is best suited for this application, however as pointed out in [3], it took many hours to implement, and is strictly suited for the task. A general purpose program is incapable of being run effectively on an FPGA. Epiphany hardware is capable of scaling with a **97% efficient rate**, and a single 64 core board ($200 all in, no other components needed) is capable of running **%73 of the performance of a 650$ desktop CPU**, taking into account of the other system component costs needed to run a desktop (In contrast, the epiphany chip (without external host board) is 75$, and the i7 desktop CPU is $350) [3].



Conclusion

The Epiphany architecture is well suited for tightly coupled memory applications due to its limited but extremely fast per core storage solution (no memory hierarchy or cache, *only register storage*). Applications which require more memory will end up stalling the pipeline and must wait more clock cycles for larger memory transfers. One solution provided a sufficient budget is by scaling up the network to provide sufficient memory as a whole on the mesh (even without the required processing power). Alternatively, if the implementation can be restructured to rely on less static memory allocations and manipulate memory transfers under the 32KB capacity per core, the Epiphany architecture would become a viable solution for great scaling and power efficiency, and of course the minimized weight and product dimensions (Great applications where an implementations weight and size and power are very important – mobile solutions). The epiphany architecture could benefit from scaling into larger core arrays and target a less hobby oriented market - If Adapteva was capable of funding this market shift. However leaving their current hobby market from which it was founded on (using the Kickstarter community), may destroy its reputation – which may be a risk worth the reward which they may need to consider in the future. It is still quite the feat of business management and engineering skills to produce 4 revisions and bring them to market with development cost restrictions they had to deal with.

By far the most unique architectural implementation in Epiphany is the Epiphany Instruction Set Architecture. By providing mixed instructions like the multiply accumulate, reducing the typical ARM instruction set, as well as supporting 16bit encoding to pack instructions with 32bit instructions; shortening its processor pipeline to decrease cycle complexity. All of these factors allow the architecture to be a refined and simple solution, whilst improving on their overall performance and achieving their design goals for power efficiency. It will certainly be interesting to see what Adapteva does in the future.

References

[1] Epiphany Architecture Reference (14.03.11 ed.). (2011). Lexington, MA: Adapteva. http://adapteva.com/docs/epiphany_arch_ref.pdf

[2] Epiphany SDK Reference (5.13.09.10 ed.). (2013). Lexington, MA: Adapteva. http://www.adapteva.com/docs/epiphany_sdk_ref.pdf

[3] Energy-efficient bcrypt cracking. (2014, August 6). OpenWall. http://www.openwall.com/presentations/Passwords14-Energy-Efficient-Cracking/

[4] Varghese, A., Edwards, B., Mitra, G., & Rendell, A. (2013). Programming the Adapteva Epiphany 64-core Network-on-chip Coprocessor. Retrieved October 3, 2015, from http://arxiv.org/pdf/1410.8772.pdf

[5] Face Detection using the Epiphany Multicore Processor. (2012, October 12). Retrieved November 9, 2015, from http://www.adapteva.com/white-papers/face-detection-using-the-epiphany-multicore-processor/