Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3

Neil Costigan, Michael Scott

School of Computing
Dublin City University
Glasnevin, Dublin 9,
Ireland.
neil.costigan@computing.dcu.ie** mike@computing.dcu.ie

Abstract. Recently the major performance chip manufacturers have turned to multi-core technology as a more cost effective alternative to ever increasing clock speeds. Well known examples of multi-core architectures include the Intel Core Duo and AMD Athlon 64 X2 range of chips. IBM have introduced the Cell Broadband Engine (Cell) as their next generation CPU to feed the insatiable appetite modern multimedia and number crunching applications have for processing power.

The Cell is the "Wicked Smart"[1] technology at the heart of Sony's Playstation 3$^{\text{TM}}$. The Cell contains a number of specialist synergistic processor units (SPUs) optimised for multimedia processing and offer a rich programming interface to applications that can make use of the vector processing capabilities. The specialised hardware design for gaming will always deliver performance gains compared to a more generic processor for its specific domain. Multi-precision number manipulation for use in cryptography is a considerable distance away from this domain. This paper explores the implementation and performance gains when using the vector processing capabilities for SSL and shows that big improvements are still possible with the hardware designed primarily for other purposes.

1 Why SSL?

Despite huge gains in computing performance and bandwidth, the widespread use of secure communications on the Internet is still essentially limited to SSL connections for password logins or with credit card payments. Despite this SSL implementations are widely distributed and well analysed for security weaknesses making it the de-facto standard for secure communications. The main reason for such limited usage is the perception that encrypted communication protocols

** Research supported by the Irish Research Council for Science, Engineering and Technology,(IRCSET).

[1] "Wicked Smart" is an advertising slogan used by Sony

such as SSL place too high demands on bandwidth and processing power at the server side of the communication and can interrupt the browsing experience of the client. This paper sets out to show that with the performance of modern multi-core hardware devices it is now possible to enable secure channels for a wider range of network communications.

2 The Cell Broadband Engine

When Sony examined the options for the Playstation 2's successor they realised that traditional clock speed improvements were not going to deliver to next generation demands. They wanted something more than the traditional CPU if the Playstation brand was going to maintain its lead over its chief competitor Microsoft's XBox$^{\text{TM}}$brand of gaming consoles. In early 2001 they turned to IBM and Toshiba. Together they formed a partnership to deliver a chip that would both provide the power for the next generation of media rich gaming consoles, while also being a scalable, adaptable design that would meet the most demanding computational tasks. The result is a unique architecture combining a traditional central processor and specialised high performance processors similar to those found in graphics cards (GPUs). These processing units are combined across a circular high bandwidth bus to offer a multi-core environment with two-instruction sets and enormous processing power. Sony use a subset of the chip inside its Playstation 3 gaming and media console. IBM offer a range of configurations inside a Blade series suitable for server and super-computing use. Central to the Cell Broadband Engine (more commonly referred to as 'Cell') is a 3.2 GHz 64-bit Power Processing Unit (PPU). The PPU is a variant (970) of the G5/PowerPC product line, a RISC driven processor found in IBM's servers and Apple's PowerMac range. This PPU works as the primary processor and as supervisor for the other cores.

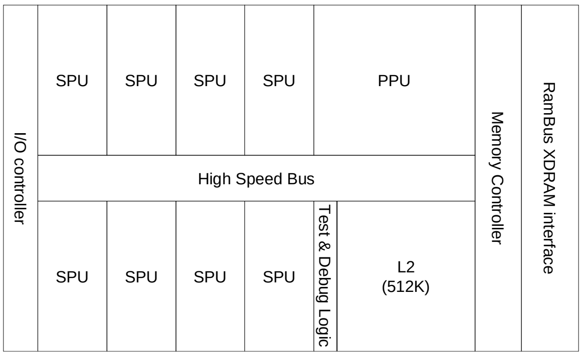

Fig. 1. Cell BE Die Layout

2.1 The Cell's SPU

The real power of the Cell is in the ability to harness the additional Synergistic Processing Units (SPUs). The SPU is a specialist processor with a RISC-like SIMD instruction set and a large (128) array of 128-bit registers. Each SPU has its own local memory store (LS). Currently, and on the Playstation 3, this LS is limited to just 256K. The SPU can access the LS in the same clock cycle as its register operations. The latest SDK (2.0) has a beta software cache implementation. While the SPU does not directly access main memory the central PPU can access each SPU's local memory store. The SPU has no hardware cache so each software application directly manages data transfer to and from each SPU. This leads to a number of interesting programming models.

While the architecture allows for any number of SPUs, a standard Cell, and those currently in production, has 8 SPUs. Interestingly Sony have chosen to utilise just 7 as they can gain much higher production yields if they can discard an SPU that shows a failure during silicon testing. Furthermore Sony restrict access to one SPU for Digital Rights Management (DRM) purposes on a Playstation running in Linux mode.

A processor with just 256K, no hardware cache and with no access to I/O doesn't appear to be anything exciting when compared to the PPU or other modern CPUs. The fact that the Cell offers 8 SPUs on one die all designed to operate in parallel combined with the ability to work with up to 4 32-bit integer operations in just one clock cycle (referred to as SIMD) that make the SPU so interesting. The SPU also contains 2 instruction pipelines and while the pipelines are not equal, careful management of the order of instructions can lead to huge amounts of data being processed with very few clock cycles and a very low clock cycles per instruction (CPI) ratio.

The large register size is ideal for the number crunching operations required for cryptography. However, the fact that the size of the register is too large for most high level language's basic types, and that most operations work with, at most, 32-bit sub-sections of the quadword register, makes development a little tricky. The programmer accesses the registers through a set of C extensions which operate exclusively on vectors rather than traditional direct memory access. The C extensions (or intrinsics) also offer a degree of code portability with similar CPUs such as the Altivec [10]. It is possible to develop small, dedicated, standalone, SPU applications (spulets). A more interesting, but more complex model, is the capability of the PPU to call SPU applications through a POSIX threads-like library passing data through a direct memory access (DMA) library.

To stimulate interest within the development community IBM offer a software development kit and ample documentation [1]. This SDK contains a full range of development tools and code samples including a powerful cycle accurate simulator for the SPU and a vector optimised multi-precision Math library (IBM MPM) [6].

2.2 Multi-instruction sets

One interesting issue with the different architectures of the PPU and SPU is the need for multi-instruction set binaries. Traditional applications compile individual source modules and then link the results to bind all program data symbols (variable, types functions etc.), but as the SPUs LS memory is physically separate and makes use of wide 128-bit registers, its program code needs to be compiled and linked separately. Both the SPU and PPU use standard ELF binary formats. An application's binary contains 64-bit code for the main PPU but embedded inside this is an object file with the SPU instructions and data ready to be pushed to the SPU on a `createthread()` call from the PPC. The build process involves two separate compilers and two linkers. The SPU ELF binary is passed through an `embedspu` command which builds a wrapper (a CESOF linkable) to the SPU binary marking it with PPU compatible symbols. Finally there is one more link stage which binds all executables together. Figure 2 [3] outlines the build process.

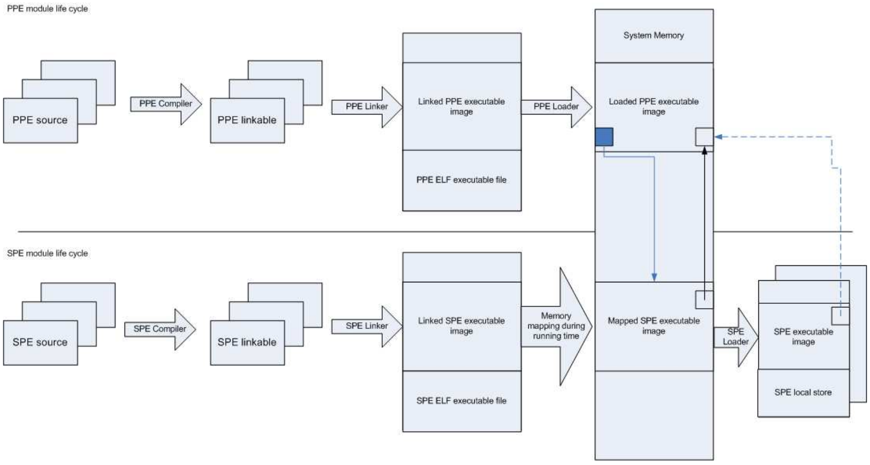

Fig. 2. Cell BE build process [3]

2.3 The Cell as a HSM

The Cell has been designed with an interesting security architecture [15]. The feature set suggest it is primarily to aid in the management of digital rights, however the interface is also open to third party developers to implement additional security functionality into code running on an SPU. This architecture can be used to make security critical code run in a protected environment as a *Hardware Security Module* (HSM). Commercially, most of the SSL accelerator vendors offer HSM's in high-end configurations.

To operate in a more protected environment the critical SSL code can be run in an SPU in *isolated mode*. For an SSL accelerator this means that is possible to have any key generation make use of a cryptographically secure random number generator, key data can be protected from other processes running on the Cell, key data can be encrypted in shared memory locations, and program code can check its integrity.

The Cell achieves this level of security by implementing a hardware based process in which

1. The code and data in an SPU can be executed in physically isolated memory space.
2. There is hardware based code signing (referred to as secure boot) where the integrity of code about to be executed can be verified.
3. An isolated process in an SPU can use hardware-key based data encryption/decryption that can only be used by code that has been verified.
4. The random number generator can be configured to use a *physical* sample source such as a Cell die. (note: this functionality is limited on our PS3 based Cell)

This extra security comes with a performance penalty. Initialisation would be affected by secure boot, if data were to be stored off SPU then there can be runtime overheads in decryption. The various random number generators degrade in performance as one moves towards a more random, physical sample based libary. Unfortunately implementation details are only available under Non-Disclosure Agreement with IBM and so we are not able to test our SPU acceleration code under this interesting environment.

For further information on the Cell see IBM's excellent Cell resource centre [5]

2.4 Direct Memory Access

As mentioned above the PPU can access main memory and has instructions to transfer data between the main memory and its registers. The SPU, on the other hand, works with its own smaller local store and so to access data from the main memory the SPU goes through a *Memory Flow Controller* (MFC) which translates SPU main memory requests over the high speed bus via a set of DMA channel calls. These DMA calls are directional (read or write), blocking or non-blocking, can be issued in parallel, and can be tagged by the programmer to allow for identification management of data.

When communicating either the PPU or an SPU can initiate and manage a DMA transfer. However it is optimal for the SPU to do the 'protocol' management as it can free PPU clock cycles that can occur if, for example, a number of SPUs have blocking calls. When the PPU needs to initiate the transfer the procedure is for the PPU to *push* a pointer to the SPU with a tag and then let the SPU *pull* the data from the pointed reference and informing the PPU, via the tag, that it has done so.

2.5 Vector Programming

To utilise the full performance of SPU SIMD instructions a developer works with a combination of Vector C extensions with assembly like code. Space is limited so we will look at the following extracts to highlight typical techniques used. We implemented a primitive `MADD()` commonly used in cryptographic libraries which fully utilises the 128-bit register by implementing a 64x64-bit multiply function.

For example the following code fragment is used to fill a quadword with two scalars (in this case standard C 64-bit `unsigned long long`) and to 'splat' across a vector. Splat is a term used when filling a vector with a mask. In a big number context we utilise splats to allow us operate on different elements of a quadword when filling partial products.

```
unsigned long long _a, _b
vector unsigned short AB;
AB=(vector unsigned short)  \
             spu_insert(_a,(vector unsigned long long)AB,0x1);
AB=(vector unsigned short)  \
             spu_insert(_b,(vector unsigned long long)AB,0x0);
/* select two bytes */
const vector unsigned char splat_short1= \
           (vector unsigned char)(VEC_SPLAT_U32(0x80800405));
```

Here we utilise a C macro to guarantee all vector multiplies (`spu_mulo()`) are at a 16-bit level to efficiently use the 16x16-bit multiplier in the SPU.

```
#define MULTIPLY(a, b)\
(spu_extract(spu_mulo((vector unsigned short)spu_promote(a,0)\
           ,(vector unsigned short)spu_promote(b, 0)),0))
```

Finally an assembly-like example of a speed up technique when adding a 128-bit value to a 64-bit value where we know there is no need to manage an overflow. This technique is used in summing partial products inside the big number multiply.

```
vector unsigned int _out_s, _in_a128, _in_a64;
vector unsigned int _sum, _c0, _t0;

_c0    = spu_genc(_in_a128, _in_a64); // generate carry bits
_sum   = spu_add(_in_a128, _in_a64); // add
_t0    = spu_slqwbyte(_c0, 4); // shift quadword left 4 bytes
_out_s  = spu_add(_sum, _t0);      // add in the carry
```

3 OpenSSL

OpenSSL [7] is an open source toolkit released under under a BSD style license. It evolved out of Eric Young's popular SSLeay and in 1998 passed to a dedicated team of developers. It has since become the de facto open source SSL toolkit. It is the security sub-system of choice for large open source projects such as Apache [8] and MySQL [13] and included in virtually all UNIX distributions including Linux, MacOSX$^{\text{TM}}$, and Solaris$^{\text{TM}}$.

The name 'OpenSSL' is misleading as the toolkit provides a vast array of building blocks and interfaces from cryptographic primitives through big number routines to PKI components such as certificate authorities and OCSP responders. One of the most useful features is the ability to factor out processing intensive operations to specialist hardware through an 'engine' interface. It is through this engine subsystem that we accelerate SSL by using the Cell SPU's vector processing capabilities.

SSL operates in two phases: an initial handshake and a bulk encryption phase. The purpose of the handshake is to swap identification credentials, algorithm capabilities, and negotiate a bulk encryption key. The reason for the key negotiation is that asymmetric cryptography, whilst needed to establish a shared secret, incurs a large computational overhead compared to a symmetric encryption algorithm. By analysing clock cycles, Zhao *et al.* [2] found that 90.4% of the SSL handshake comprises public key operations. Cryptographic operations take, in total, about 95% of the total CPU load.

Since the CPU load will be heaviest at the server side, and since the main computationally load incurred by the server for its part in the handshake is asymmetric decryption, we focus our attempts on speeding up asymmetric decryption.

Isolating the SSL handshake to measure our improvements is a challenging task as there are can be many dependencies (network traffic, HTTP server etc.) on a running machine which make accurate sampling difficult. Fortunately OpenSSL provides the utility `openssl speed` which can measure individual algorithms. Using this utility we can demonstrate improvements to the throughput of the critical algorithms. The SSL protocol supports a range of asymmetric algorithms, (RSA, DSA, ECC etc.). In this paper we focus on RSA but the technique is relevant to all.

3.1 OpenSSL engine

When taking over computational tasks from OpenSSL two issues which must be considered are

1. How the engine informs the library of the scope of its responsibilities.
2. Marshalling the big number format to and from OpenSSL's internal representation.

To tell OpenSSL exactly what the engine will do the developer provides a static library with a set of defined interfaces with descriptive text to describe the engine. Then, through function pointer replacement, a defining a set of functions which implement the algorithms that the engine intends to provide.

While there are dynamic loading techniques for closed source libraries, at the current OpenSSL version (0.9.8d) the simplest method to integrate the engine is to statically link the engine code and add a call inside ENGINE_load_built_in_engines(). This will add the engine as an option to any application using the OpenSSL default engine.

Through this call OpenSSL then loads any engine that conforms to the correct interface at start-up, and subsequently any OpenSSL command that uses the -engine <id> option will redirect to the named engine. At the Engine_init() stage the calling library passes an OpenSSL data structure containing a set of initialisation variables and an opening via a free additional pointer for the engine to append its own data structure which can be accessed later by subsequent engine functions. The engine is responsible for its own memory management. It is through this Engine_init() call that we gather OpenSSL parameters and convert the OpenSSL big number representation to the native Cell IBM Multiprecision Big number format.

4 Development

To recap: we need to build a PPU library (32 or 64-bit) that plugs into a PPU build of OpenSSL. Inside this library we embed an SPU ELF executable which can act upon the 128-bit registers and utilises IBM's MPM library. This SPU ELF executable needs to be under 256K including all code and data. The multicore environment with the limitations on code and data size requires some unconventional, data centric, programming models which the engineering community are still evolving. The cardinal rule appears to be 'offload as much as one can to the SPUs'. Many data intensive multimedia applications employ a model where data is streamed through a chain of SPUs with each SPU carrying out a specific operation on the data, then calling another SPU with the processed data. Yet another model makes the PPU act as a scheduler pushing data segments and code 'blobs' to any SPU with the PPU managing the operations and data ordering through double buffering. To fit the OpenSSL engine model, we mirror the operation of a similar engine developed by Geoff Thorpe of the OpenSSL core team for the GNU Multi-Precision library (GMP) [9]. To have the SPUs do as much work as possible we chose to overload the RSA_mod_exp() function and indicate through control flags that the engine would perform full RSA decryption using the Chinese Remainder Theorem. Figure 3 describes the interaction between the various components. This allows us to potentially parallelise the modular exponentiation calls. We could approach this a number of ways:

1. Have the PPU do the RSA/CRT but invoke SPUs to manage the expensive modular exponential (mod_exp()). Different SPUs would handle the p and q mod_exp().
2. Have the PPU pass the whole RSA/CRT to an SPU.
3. Have the PPU pass the whole RSA/CRT to an SPU with this SPU passing the the two mod_exp() to two other SPUs.
4. Have the PPU pass the whole RSA/CRT to an SPU with this SPU passing one of the two mod_exp() to another SPU and, in parallel, handle the other.

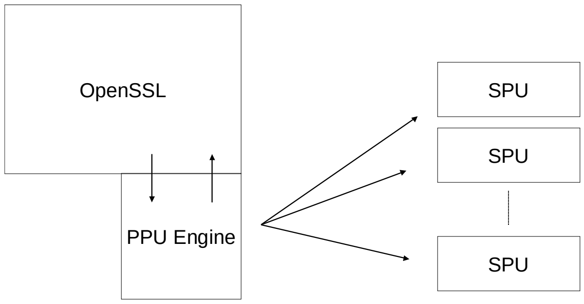

Fig. 3. OpenSSL with Engine and SPUs

There are a number of advantages to each. With (1) the amount of data in the DMA bus is reduced but it breaks the guideline of offloading as much computation as possible to an SPU. With (3&4) the latency per SSL connection will be reduced but, as it adds extra DMA data to the bus, the over all maximum throughput will be affected. With (2, 3 & 4) we can double buffer the data transfer, for example passing the p parameter to the bus while the SPU is processing the q `mod_exp()`. The double buffering technique would offer relatively small speed gains. We implemented (1) and (2) and found the initial speed up to be marginally higher but the maximum throughput to be slightly lower. This is explained by increased amount of SPU invocation. In an attempt to measure the maximum throughput we chose to focus on (2).

To maintain compatibility with OpenSSL and other engine implementations we use notation matching OpenSSL code: dmp_1, the decryption exponent mod $p-1$, dmq_1, the decryption exponent $\mod q-1$. $iqmp$ is the inverse of q mod p. I_0 is the cyphertext. A decryption exponent d, for a prime p, is a number d, such that $m^{ed} \mod p = m$ or $ed = 1 \mod (p-1)$, where e is the encryption exponent, commonly chosen to be 3 or 65537.

At the RSA initialisation stage OpenSSL passes the $(p, q, dmp_1, dmq_1, iqmp)$ parameters to the engine. At this stage we check the parameters, allocate a memory store, fill the store with local copies of the big numbers ready to pass to an SPU, and then pass the memory store pointer back through a thread safe thread local memory store. OpenSSL later makes a call to the main overloaded `RSA_mod_exp()` function with I_0 and the same thread memory store parameter. The overloaded `mod_exp()` extracts the thread local data, calls an SPU thread, DMA transfers the location and size of the memory store to the SPU. It then allocates space for the return data from the SPU.

As mentioned above, the SPU thread when activated could either receive all parameters in a full DMA transfer or, more efficiently, a pointer to the block of big numbers in memory on the Cell's main store. By passing the pointer, the SPU's memory flow controller effectively takes the memory processing away from the main PPU, further improving the performance.

At this stage the SPU thread converts the big number set to the IBM MPM format and carries out the CRT logic. On success it takes the result, pushes it back to the PPU using the DMA tag that the I_0 parameter was sent with, finally cleaning up any memory used by the engine and exiting.

4.1 RSA/CRT

We implement traditional RSA Decryption using Chinese Remainder Theorem but with a small modification. Because the SPU is restrictive in some respects and as we can't be certain that the parameter p is always greater than q we need to maintain a sequence of calls that ensure the results of any modular exponentiation stay positive.

1. The SPU compiler optimiser is most efficient when there is no branching.
2. The current version (SDK 2.0) of the IBM MPM is intended to work with unsigned numbers.
3. Integer comparison operations (less than, greater than) on negative numbers are undefined.

To overcome these restrictions we assume p is always less than q. A condition OpenSSL guarantees. The modified algorithm is outlined in Algorithm 1.

Algorithm 1 RSA Decryption using Chinese Remainder Theorem modified for the IBM MPM unsigned restrictions. *Note: We follow OpenSSL notation found in all engine implementations.*

```
INPUT: $p, q, I_0, dmq_1, dmp_1, iqmp$
OUTPUT: $r_0$

  $r_1 \leftarrow I_0 \mod q$
  $m_1 \leftarrow {r_1}^{dmq_1} \mod q$
  $r_1 \leftarrow I_0 \mod p$
  $r_0 \leftarrow r_1^{dmp_1} \mod p$
  $r_0 \leftarrow r_0 - m_1$
  if $r_0 < 0$ then
      $r_0 \leftarrow r_0 + p$
  end if
  $r_1 \leftarrow r_0 \cdot iqmp$
  $r_0 \leftarrow r_1 \mod p$
  $r_1 \leftarrow r_0 \cdot q$
  $r_0 \leftarrow r_1 + m_1$
```

The IBM-MPM library offers an alternative modular exponentiation function which uses the Montgomery reduction technique. This is more efficient than the classic 'product then reduce the result modulo n' approach as it keeps the numbers from growing unnecessarily. See chapter 14 of [12].

5 Results

Table 1 lists timings in cycles counts and milliseconds for the time consuming functions of the RSA/CRT implementation. Two totals are presented: sum of these calls and an observed timing for all calls including some initialisation and the DMA receive calls. These timings are made using an engine with just one SPU configured.

function	calls	cycle count	cycles	millisecs	secs	#/sec
big_number_ convert()	7	877	6139	0.00192		
mpm_mod()	4	77731	310924	0.09716		
mpm_mont_mod_exp()	2	93328909	186657818	58.33057		
mpm_mul()	1	22733	22733	0.00710		
mpm_sub()	1	704	704	0.00022		
mpm_add()	1	1116	1116	0.00035		
mpm_madd()	1	39648	39648	0.01239		
sub-total		187039082			0.05845	
Total *includes other calls*		215159632			0.06724	14.87

Table 1. RSA/CRT decryption implemented in IBM MPM function calls with cycle count and time in milliseconds for a 4096-bit key

We can see that, as expected, the `mpm_mont_mod_exp()`[2] calls represent the bulk of the time consuming operations. A case could be made for a design that offloaded just this call to an SPU. Theoretically (from the results of Table 1) we can expect the SPU to be able to process 14.8 4096-bit decryptions in a second. Interesting (from Table 2) we achieve close to this at 14.1. Obviously there is additional overhead from DMA and the process queue on the main PPU. Cycle counts are from the latest (2.0) version of the SDK's simulator. Unfortunately the simulator (at this time) cannot measure DMA or PPU latency.

As mentioned previously, to get some sense of the improvements our optimisations have made we use the `openssl speed` command on RSA with the engine off (native OpenSSL on the PPU) and with our engine on utilising the SPU.

Tests are run on a 3.2 GHz Playstation 3 with just **6** SPUs running Yellow Dog Linux 5.0 [16] with kernel version 2.6.16-20061110.ydl.1ps3. A server/blade Cell system would have up to 16 SPUs. We could expect the Playstation Cell to deliver a throughput of up to 89 sign/sec and a blade server to go as high as 237 sign/sec. Our observations (Table 3) see slightly smaller results. As mentioned there are number of factors that could skew our observed numbers, mainly the design of the OpenSSL speed post-processing, DMA overhead and the fact that the PPU is busy managing the multiprocess queue.

[2] The generic `mpm_mod_exp()` clocks at 136914856 cycles for a 4096-bit modulus

We are using the `openssl speed -elapsed` time option instead of the more often quoted CPU user time as on the multi-core processor the CPU timer will just count the CPU time of the driving PPU thread whereas the multi-threaded nature of the SPU based system is better represented by elapsed time. OpenSSL is configured for 64-bit PPC/G5 ASM [3].

```
openssl speed rsa −elapsed
openssl speed rsa −engine cellspumpm −elapsed
```

RSA	PPU		1 SPU	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.003435s	291.2	0.005655s	176.8
2048-bits	0.017541s	57.0	0.015636s	64.0
4096-bits	0.109793s	9.1	0.070915s	14.1

Table 2. OpenSSL speed on PPU vs. 1 SPU using IBM-MPM on 3.2GHz Cell

From Table 3 we can see that the overhead of the DMA transfer and the big number conversion impact the performance improvements just below the 2048-bit key. The benefits of the 128-bit registers are apparent at 4096-bit level with improvements in the order of 150% (14.1 vs. 9.1).

To see the full impact of the multi-core we need to use the -multi [n] option to the speed command which can (through `fork()`) generate multiple simultaneous RSA operations. We have picked a number (6) of parallel processes to run matching the number of SPUs on the Playstation 3. It is important to note that the -multi option introduces some small processing overhead to the speed command as it uses a fork() invocation whereas the standard calls in single threaded. Again we compare the PPU with an SPU enabled engine.

```
openssl speed rsa −elapsed −multi 6
openssl speed rsa −engine cellspumpm −elapsed −multi 6
```

We see from Table 3 similar overheads impacting the 1024-bit keys. However there is huge improvements in 2048-bit (329.7 vs.71.7) and 4096-bit (83.6 vs 11.2). A 749% increase.

While the `openssl speed` utility running on the 6 SPU Cell inside a Playstation 3 gives us a solid basis to develop and measure our improvements, Séan Starke at IBM was kind enough to try our tests in a full 16 SPU dual Cell blade. These results (Table 4) are preliminary but are consistent with the trend from the Playstation 3 results.

[3] Options: bn(64,64) md2(int) rc4(ptr,char) des(idx,risc1,16,long) aes(partial) idea(int) blowfish(idx) compiler: ppu-gcc -DOPENSSL_USE_MPM_SPU -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -m64 -DB_ENDIAN -DTERMIO -O3 -Wall

RSA	PPU		6 SPUs	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.000724s	384.5	0.001906s	524.7
2048-bits	0.002600s	71.7	0.003033s	329.7
4096-bits	0.089455s	11.2	0.011925s	83.9

Table 3. OpenSSL speed on PPU vs. 6 SPUs using IBM-MPM on 3.2GHz Cell, 6 parallel processes.

RSA	2 PPUs		16 SPUs	
key length	sign	sign/sec	sign	sign/sec
1024-bits	0.001270s	787.5	0.001509s	662.7
2048-bits	0.006805s	146.9	0.001664s	601.0
4096-bits	0.043944s	22.8	0.005762s	173.6

Table 4. OpenSSL speed on 2 PPUs vs. 16 SPUs using IBM-MPM on 3.2GHz Cell, 16 parallel processes.

6 Conclusions and Future Work

The numbers speak for themselves: over 700% improvement in performance. With the widespread use of specialised multi-core processors there is no reason to prevent the roll out of always on encryption leading to improvements in privacy for the general user.

We believe that we have pushed the Cell SDK's IBM-MPM library to its limits. The library is an excellent demonstration of the power of SPU intrinsics 'vector' programming. However, we believe the introduction of an optimised number library more suited to crypto can substantially improve the performance, possibly doubling the figures presented above.

As mentioned the results are based on using generic Montgomery `mpm_mont_mod_exp()` function. This function allows for any size of parameter whereas we know the size of parameters are based on fixed key lengths (1024, 2048 etc.) These fixed lengths can offer further optimisations as they always align on the 128-bit boundaries of the vectors and that the number of partial products to be summed inside any multiplies can be determined allowing for very efficient carry management.

The multiplication inside the `mpm_mont_mod_exp()` needs to be examined in more detail. MPM uses 'row by row' operand scanning to do big number multiplies whereas a 'column by column' product scanning technique used by the Comba [4] method would be more suitable for the large, fixed sized numbers used by crypto. Furthermore, as the number length moves beyond 1024-bit the Comba method can be combined with the Karatsuba technique [11] for further improvement.

OpenSSL uses this Comba/Karatsuba combination at key lengths above 1024-bit irrespective of the architecture. We hope to swap out the IBM MPM library and use a fine tuned version of MIRACL [14] with fixed key sizes on fixed 128-bit alignment, and utilising the Comba/Karatsuba speed ups on longer key lengths.

The threshold key length to optimally use the Karatsuba method depends heavily on the underlying word size and the architecture's instruction set, specifically how fast the multiplier is compared to the addition. We hope to examine this threshold in more detail with the more flexible, MIRACL library.

The performance figures focus on raw crypto performance. We would like to examine the SSL performance of real word data using a commercial grade SSL/HTTP load testing suite. We also intend to offer support for DSA and ECC algorithms.

7 Acknowledgements

For development tools and background information we turned again and again to the IBM's 'DeveloperWorks' resource centre and the Cell SDK. We would like to thank the Cell development community particularly Séan Starke and the IBM team. The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research. We would also like to acknowledge the valuable feedback given by the anonymous reviewers from the SPEED 2007 (Amsterdam) workshop at which this paper was presented. Finally We would also like to thank Augusto Jun Devegili (Unicamp, Brazil), Peter Kehoe (DCU), Noel McCullagh, and Stephen Henson (OpenSSL)[7] for their encouragement, assistance & patience.

References

1. IBM alphaWorks. Cell Broadband Engine SDK. `http://www.alphaworks.ibm.com/topics/cell`.
2. Zhao Iyer Srihari Makineni Laxmi Bhuyan. Anatomy and performance of SSL processing. In *Proc. IEEE Int. Symp. Performance Analysis of Systems and Software*, pages 197–206, 2005.
3. Alex Chunghen Chow. Programming the Cell Broadband Engine. *Embedded Systems Design*, 2006. `http://www.embedded.com/columns/showArticle.jhtml?articleID=188101999`.
4. P. G. Comba. Exponentiation cryptosystems on the ibm pc. *IBM Syst. J.*, 29(4):526–538, 1990.
5. IBM DeveloperWorks. Cell Broadband Engine resource center. `http://www-128.ibm.com/developerworks/power/cell/`.
6. IBM DeveloperWorks. Cell Broadband Engine SDK Libaries Multi-Precision Math Library, 2006. `http://www-128.ibm.com/developerworks/power/cell/`.
7. S. Henson et al. OpenSSL library. Open source library, 1988. `http://www.openssl.org`.

8. Apache Foundation. Apache HTTP server project. http://www.apache.org.
9. SWOX / Free Software Foundation. GNU Multiple Precision Arithmetic Library. http://www.swox.com/gmp/.
10. Freescale. Altivec velocity engine. http://www.freescale.com/altivec.
11. Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.
12. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
13. MySQL. MySQL open souce database. http://www.mysql.com.
14. M. Scott. MIRACL. http://www.shamus.ie.
15. K. Shimizu, H. P. Hofstee, and J. S. Liberty. Cell Broadband Engine processor vault security architecture. *IBM Journal of Research and Development*, 51(5):521–528, September 2007.
16. Terrasoft. Yellow Dog Linux. http://www.terrasoftsolutions.com/products/ydl/.