Accelerating SSL using the Vector processors in IBM's Cell Broadband Engine

Neil Costigan, Michael Scott

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

Abstract. CUT FOR SHORT PAPER 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.

Introduction sections cut for extended abstract. There is additional editing in the following sections

1 OpenSSL

Edited short for extended abstract

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.

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

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.

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).

Edited short for extended abstract

1.1 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 **??**.

Edited short for extended abstract

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 [2].

2 Results

Edited short for extended abstract

We can see that, as expected, the `mpm_mont_mod_exp()`[1] 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 **??**) we can expect the SPU to be able to process 14.8 4096-bit decryptions in a second. Interesting (from Table 1) 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 [3] 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 2) 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.

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

From Table 2 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.

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

We see from Table 2 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.

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

Edited short for extended abstract

3 Conclusions and Future Work

Edited short for extended abstract

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.

4 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. 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 workshop at which this paper was presented.

References

1. S. Henson et al. OpenSSL library. Open source library, 1988. `http://www.openssl.org`.
2. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.
3. Terrasoft. Yellow Dog Linux. `http://www.terrasoftsolutions.com/products/ydl/`.