FFT Specific Compilation on IBM Blue Gene

Stefan Kral

Anzengrubergasse 61/1/6
A-2380 Perchtoldsdorf, Österreich/Austria
skral@complang.tuwien.ac.at

Abstract: Bei vielen numerischen Codes gelingt es verfügbaren Compilern nicht, das Leistungspotential moderner Prozessoren zufriedenstellend auszuschöpfen. Als Alternative zum Hand-Coding und -Tuning von numerischen Grundroutinen wurde der MAP Special-Purpose-Compiler entwickelt und speziell an die Anforderungen von Codes aus der Domäne der Signalverarbeitung angepaßt. Die neue, an IBM Blue Gene Supercomputer adaptierte Variante erreicht bei der Übersetzung von Codes der marktführenden Programmbibliothek FFTW bis zu 80% der Maximalleistung der in IBM Blue Gene Systemen verwendeten IBM PowerPC 440 FP2 Prozessoren. Der MAP Compiler trägt so wesentlich zur Steigerung der Leistung von naturwissenschaftlichen Anwendungsprogrammen auf IBM Blue Gene, wie z. B. der materialwissenschaftlichen Simulation QBOX, dem Gewinner des ACM Gordon-Bell-Preises 2006 in der Kategorie "Peak Performance", bei.

1 Einführung

Algorithmen zur digitalen Transformation von Signalen sind im Scientific-Computing von größter Bedeutung und werden in zahlreichen Gebieten angewendet – von der Echtzeit-Signalverarbeitung in eingebetteten Systemen bis zur numerischen Lösung partieller Differentialgleichungen im Rahmen komplexer Simulationen auf Supercomputern.

Die Entwicklung und Publikation der schnellen Fourier-Transformation (FFT) durch Cooley und Tukey [CT65] im Jahr 1965 leitete die Entwicklung einer neuen Klasse schneller Signalverarbeitungsalgorithmen ein. Diese haben – im Gegensatz zur direkten Auswertung des entsprechenden Matrix-Vektor-Produktes – nicht eine Berechnungskomplexität von $O(N^2)$, sondern von nur $O(N \log N)$.

Für viele Transformationen gibt es jedoch nicht *einen* eindeutig bestimmten, sondern eine Vielzahl äquivalenter $O(N \log N)$-Algorithmen, die sich nur wenig bezüglich ihres Rechenaufwandes unterscheiden.

Daß sie bezüglich ihrer Speicherzugriffsmuster stark voneinander abweichen, bekam erst im letzten Jahrzehnt durch die weite Verbreitung mehrstufiger Speicherhierarchien – in Form von Caches – Bedeutung: War bis dahin die Reduktion der Anzahl arithmetischer Operationen vorrangig, so dominierten fortan die Kosten für Speicherzugriffe die Gesamtkosten. Diese Zeit eines neuen Softwareparadigmas numerischer Software, des Automatic-Performance-Tuning [WPD01], hatte begonnen.

Führende Automatic-Performance-Tuning-Software auf dem Gebiet der Signaltransformationen – wie z. B. SPIRAL [PMJ+05, MJJ+98] oder das mit dem Wilkinson-Preis ausgezeichnete FFTW [FJ05, FJ98] – gingen in zweierlei Hinsicht neue Wege: (i) Beide Systeme verwenden domänenspezifische Programmgeneratoren – FFTW's `genfft` [Fri99, FK01] für FFT-Grundroutinen oder SPIRAL's SPL [XJJP01] für im Tensorformalismus repräsentierbare lineare Transformationen – zur automatischen Erzeugung der benötigten numerischen Grundroutinen. (ii) Sie führen vor dem eigentlichen Lösungsprozeß eine explizite, pro Problem einmalige Planungsphase durch, um den effizientesten Algorithmus – bzw. die beste Kombination von Algorithmen – zur Lösung eines bestimmten Problems auf einer gegebenen Zielplatform zu ermitteln. Die automatische Suche nach den besten Algorithmen und Implementierungen orientiert sich dabei nicht an einzelnen quantitativen Maßen, die nur einzelne Aspekte der Ausführung auf dem Zielsystem charakterisieren, sondern basiert ausschließlich auf konkreten Laufzeitmessungen der jeweiligen Kombination von Basisalgorithmen auf dem Zielsystem.

Um Portabilität zu erreichen, erzeugen die Programmgeneratoren dieser Systeme C-Code. In Experimenten hat sich jedoch herausgestellt, daß die von verfügbaren Compilern erzeugten Assembler-Codes hinsichtlich ihrer Qualität handgeschriebenen Codes unterlegen sind. Um die Leistungslücke zwischen handgeschriebenen und maschingeneriertem Assembler-Code zu schließen wurde im Jahr 2000 das Special-Purpose-Compiler-Projekt MAP initiiert. Anders als General-Purpose-Compiler nutzt der MAP Compiler domänenspezifisches Wissen, konzentriert sich auf die in dieser Domäne wichtigsten Aspekte der Codegenerierung, und erzeugt so Assembler-Code bisher unerreichter Güte.

Für meine Dissertation habe ich – aufbauend auf die Erfahrungen eines im Rahmen meiner Diplomarbeit entwickelten Intel x86-spezifischen Compilers [KFLU03, KFL+04] – ein Redesign sowie eine Reimplementierung des MAP Compilers für IBM PowerPC 440 FP2 Prozessoren vorgenommen und so die Qualität des erzeugten Codes wesentlich steigern können: FFTW-Grundroutinen, die mit Hilfe der Blue-Gene-Version des MAP-Compilers übersetzt werden, erreichen eine Effizienz von bis zu 80% und damit die dreifache Leistung jener Objekt-Codes, die von der aktuellsten Version des optimierenden IBM XL C Compilers erzeugt werden. Eine Variante von FFTW, deren Grundroutinen durch den MAP Compiler übersetzt wurden, "FFTW-GEL for Blue Gene" ist Teil der Blue Gene Systemsoftware geworden und stellt eine wichtigste, leistungssteigernde Komponente des materialwissenschaftlichen Simulationsprogrammes QBOX [GDS+06], dem Gewinner des ACM Gordon-Bell-Preises 2006 in der Kategorie "Peak Performance", dar.

Gliederung. Die folgenden Abschnitte gehen auf verschiedene Aspekte des MAP Compilers für IBM Blue Gene [KTU06, LKFU05, FKLU05] genauer ein. Anfangs werden die Designprinzipien und -ziele des Compilers darstellt. Dann werden die Struktur und die einzelnen Komponenten vorgestellt: Der *MAP Vectorizer* extrahiert 2-weg Parallelismus im SIMD-Stil (Single-Instruktion Multiple-Data) in numerischen Straight-Line-Codes. Der *MAP Optimizer* führt lokale Codeverbesserungen durch, wie sie üblicherweise von versierten Assembler-Programmierern manuell ausgeführt werden. Das *MAP Backend* übersetzt optimierten High-Level-SIMD-Vektor-Code in Assembler-Code für den Zielprozessor IBM PowerPC 440 FP2. Schließlich werden Messungen von verschiedenen vom MAP Compiler übersetzten FFT-Codes auf einem IBM Blue Gene System präsentiert.

2 Design des MAP Compilers

Beim Design des Compilers habe ich auf folgende Punkte besonderen Wert gelegt: (i) Er sollte so einfach wie möglich gehalten sein, sich also auf die wesentlichsten Aspekte der Übersetzung konzentrieren und für die Handhabung der allgemeinsten Fälle mit General-Purpose-Compilern kooperieren. (ii) Die Verwendung unabhängiger Komponenten sollte die Fehlersuche erleichtern. (iii) Für die Transparenz und Nachvollziehbarkeit der einzelnen Übersetzungsschritte sollte eine einheitliche, vom Menschen lesbare interne Zwischendarstellung gewählt werden.

Struktur des Compilers. Der MAP Compiler ist in Form einer offenen Toolchain realisiert und übersetzt Input-Code – in einer einfachen domänenspezifischen Sprache formuliert – schrittweise in zielspezifischen optimierten Assembler-Code. Den Eingangspunkt der Toolchain bildet der MAP Vectorizer, der in skalaren Codes inhärenten 2-weg SIMD-Parallelismus sucht und explizit macht. Der Code wird dann vom MAP Optimizer verbessert und schließlich vom MAP Backend in Assembler-Code transformiert.

Generische Komponenten/Anordnung von Komponenten. Durch konsequente Parametrisierung sind die Teile des Backends, die Ressourcenallokation sowie Instruktionsanordnung durchführen, leicht anpassbar und lassen sich innerhalb der Toolchain verschieben – etwa um mit verschiedenen Anordnungen von Übersetzungsphasen zu experimentieren.

Zwischendarstellung von Code. Für die Repräsentation von Code in seinen verschiedlichen Übersetzungsstufen verwenden alle Komponenten der Toolchain *eine* einheitliche, klartextliche Darstellung in Form von variablenfreien Grundtermen in Syntax der logikorientierten Programmiersprache PROLOG. Prozedurdefinitionen beinhalten den Prozedurnamen, die Argumentreihenfolge sowie -typen, den Code in Static-Single-Assignment Form (SSA) [App98] und verschiedene Hilfsdatenstrukturen, die fallweise eine enge Kopplung von in der Toolchain benachbarten Komponenten ermöglichen.

Inspektion/Injektion von Code. Da alle Komponenten ausschließlich über relativ enge Schnittstellen kommunizieren und die Zwischendarstellung unmittelbar in vom Menschen lesbarer Form ist, ist es leicht, nachzuvollziehen, was während einer bestimmten Übersetzungsphase *tatsächlich* passiert. Um etwaige Fehler im Compiler rasch lokalisieren zu können, kann man manuell in den Übersetzungsprozeß einzugreifen und die Übersetzung an einem beliebigen Punkt fortführen.

Annotation von Instruktionen. High-level Informationen – z. B. über Aliasing von Zeigern – können durch Annotationen an Instruktionen gebunden werden. So können noch im letzten Übersetzungsschritt aggressive, akkurate Optimierungen durchgeführt werden.

Handhabung von Kontrollfluß. Der Compiler unterstützt ausschließlich die Übersetzung einzelner Basic-Blocks – flache Codestücke ohne Kontrollstrukturen mit genau einem Eintrittspunkt und einem Austrittspunkt. Für die Unterstützung numerischer Codes mit komplexem Kontrollfluß werden Basic-Blocks in separate Prozeduren ausgelagert und mit dem MAP Compiler übersetzt. Die verbleibende Rahmenprozedur ruft die numerischen Kernprozeduren auf und wird mit einem generellen C-Compiler übersetzt.

Implementierungssprachen. Die Implementierung nutzt spezielle Merkmale verschiedenen Programmiersprachen: Komponenten mit heuristischer Suche – der MAP Vectorizer und die Befehlsauswahl zur Erzeugung von Adresscode – verwenden MERCURY, eine statisch typisierte, logikorientierte Programmiersprache, die Backtracking [GB65] effizient realisiert und als Sprachmittel anbietet. Teile des Backends – Registerallokator und diverse Scheduler – sind in CAML, einer statisch typisierten, funktionalen Programmiersprache realisiert. Die dynamisch typisierte logikorientierten Programmiersprache PROLOGwurde für die rasche Entwicklung von internen Prototypen und für "Glue-Code" eingesetzt.

2.1 Zuständigkeit der einzelnen Komponenten für Optimierungen

Da Einfachheit wohl das wichtigste Designziel des Compiler war, sind die Verantwortlichkeiten bezüglich Code-Optimierung klar verteilt.

Speicherzugriffsmuster. Automatic-Performance-Tuning Systeme optimieren die Reihenfolge von Speicherzugriffen auf hohem, problemspezifischen Niveau. Die Komponenten der Toolchain sind daher bestrebt, die durch die Programmgeneratoren dieser Systeme vorgegebenen Speicherzugriffsmuster zu erhalten.

Instruktionsauswahl. Die Auswahl von Gleitpunkt-Recheninstruktionen erfolgt im MAP Vectorizer und im MAP Optimizer: Der MAP Vectorizer extrahiert Parallelismus im SIMD-Vektor Stil. Der MAP Optimizer ist für das Bilden von Fused-Multiply-Add (FMA) Instruktionen sowie für die Optimierung SIMD-spezifischer Instruktionsmuster zuständig. Das MAP Backend wählt Ganzzahl-Instruktionen für die Berechnung effektiver Adressen aus und fügt die für die Einhaltung der Prozedur-Aufrufkonvention sowie für Stackzugriffe benötigten Speicherzugriffsinstruktionen in den Code ein.

3 Implementierung des MAP Compilers

Im Folgenden werden die drei Kern-Komponenten des MAP Compilers – MAP Vectorizer, MAP Optimizer und MAP Backend – beschrieben.

3.1 MAP Vectorizer

Viele moderne Prozessoren – wie auch der in IBM Blue Gene [MAA+05] verwendete IBM PowerPC 440 FP2 – bieten SIMD-Instruktionssatzerweiterungen [Doc01], die das Potential haben, numerische Anwendung wesentlich zu beschleunigen. Anders als klassische SIMD-Vektorcomputer arbeiten diese neue SIMD Erweiterungen mit kurzen Vektoren fixer Länge. Aktuelle Zielarchitekturen unterstützen typischerweise zwei Gleitpunktzahlen doppelter Genauigkeit oder vier Gleitpunktzahlen einfacher Genauigkeit pro Vektor.

Diese modernen SIMD-Erweiterungen wurden ursprünglich für die Beschleunigung von Multimedia-Anwendungen entworfen und erlauben, Parallelismus auf sehr niedriger Ebene auszudrücken. Daher ermöglichen sie nicht nur den Einsatz schleifenbasierte Vektorisierungsmethoden für Vektorcomputer [ZC91], sondern auch die Vektorisierung auf der Ebene einzelner Basic-Blocks [LA00, LB00], was sich günstig auf die Codegröße und den Registerdruck auswirken kann.

Der MAP Vectorizer versucht – unter Verwendung heuristischer Suche – skalare Basic-Blocks mit 2-weg SIMD-Instruktionen, die von der Zielarchitektur unterstützt werden, bestmöglich zu überdecken und zugleich die Zahl der benötigten SIMD-Umordnungsinstruktionen gering zu halten. Abbildung 1 zeigt anhand eines kleinen Beispiels die Eingabe und Ausgabe der Vektorisierung. Durch Anpassung des Vectorizers an Codes der Domäne der Signaltransformation ist es gelungen, erfolgreich zahlreiche Codes – darunter reelle FFT Grundroutinen – zu vektorisieren.

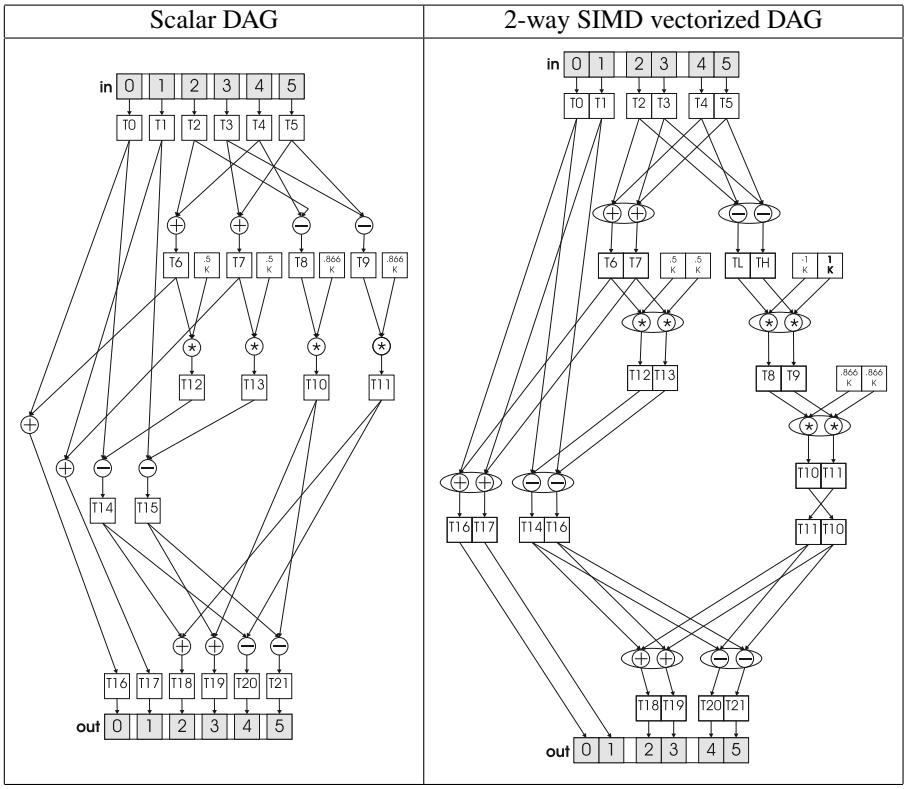

Abbildung 1: **Vektorisierung eines skalaren komplexen FFT-Codes der Größe 3.** Der skalare Graph auf der linken Seite ist berechnungsäquivalent mit dem SIMD-Graphen auf der rechten Seite. Eine SIMD-Multiplikation mit der Konstante $(-1, 1)$ mußte eingefügt werden, um das Vorzeichen *eines* Teiles eines Vektors zu ändern. Der SIMD-Code auf der rechten Seite, der die Ausgabe des MAP Vectorizers ist, ist in weiterer Folge die Eingabe des MAP Optimizers.

3.2 MAP Optimizer

Der MAP Optimizer besteht aus einem flexiblen committed-choice Termersetzungssystem und einer Menge von Ersetzungsregeln, die an die Zielarchitektur und die Problemdomäne angepaßt sind. Durch die regelbasierte Transformation bestimmter, durch Datenabhängigkeiten verbundener Instruktionssequenzen in gleichwertige, effizientere Varianten verbessert der Optimizer die lokale Codequalität und bewahrt die durch den Vectorizer vorgegebene globale Codestruktur. Die Unterstützung unterschiedlicher Prioritäten für die einzelnen Regeln erleichtert die Entwicklung und Wartung größerer Umschreibungssysteme und hält gleichzeitig den Optimierungsprozeß nachvollbar.

Die Regelbasis des MAP Optimizer für IBM Blue Gene besteht aus zwei Arten von Regeln: Die eine sorgt unmittelbar für eine Verbesserung der Codequalität – durch die Ausnutzung von SIMD-FMA-Instruktionen oder durch das Verkürzen von Pfadlängen in dem gerichteten azyklischen Graphen (GAG), der dem Input-Code entspricht. Die andere führt nicht unmittelbar zu einer Verbesserung, sondern versucht, weitere Regelanwendungen zu ermöglichen – z. B. durch Verschieben von SIMD-Vektor Umordnungsinstruktionen oder Multiplikationen mit numerischen Konstanten innerhalb des GAG, wie in Abbildung 2 dargestellt.

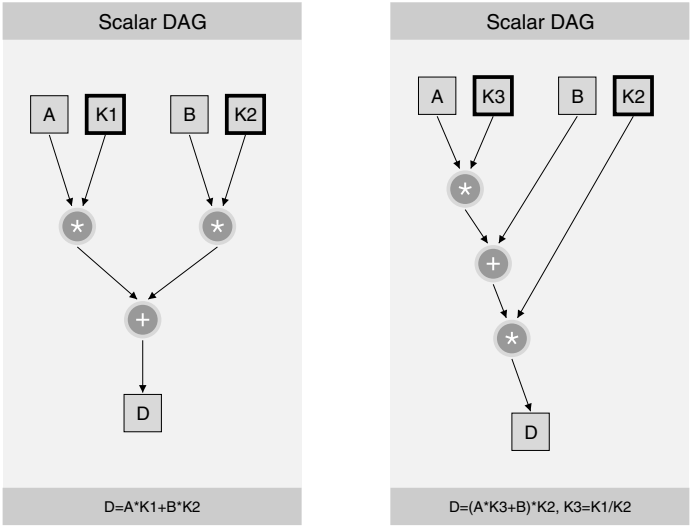

Abbildung 2: **Bilden von skalaren FMA-Instruktionen.** Der Graph auf der linken Seite zeigt ein in FFT-Codes häufig vorkommendes Instruktionsmuster, das aus zwei Multiplikationen mit Konstanten und einer abhängigen Addition besteht. Dieses Muster, das unmittelbar auf eine Multiplikation mit einer abhängigen FMA-Instruktion abgebildet werden kann, wird in ein gleichwertiges, das im Graphen auf der rechten Seite dargestellt ist, transformiert. Durch Propagieren der Multiplikation in Richtung D kann sie in weiteren Ersetzungsschritten potentiell mit abhängigen Instruktionen – wie Additionen oder Multiplikationen – zusammengelegt werden. Analog zu skalaren FMA-Instruktionen werden auch SIMD-FMA-Instruktionen gebildet.

3.3 MAP Backend

Anders als Vectorizer und Optimizer ist das MAP Backend als eine Reihe kleinerer Module implementiert, die Ressourcenallokation (für Ganzzahl- sowie Gleitpunkt-Register) und verschiedene Ebenen von Code-Anordnung realisieren.

Berechnung effektiver Adressen. Bei vielen Signalverarbeitungscodes ist in der Praxis die Zahl der Gleitpunkt-Rechenoperationen von derselben Größenordnung wie die Gesamtanzahl aller Lade- bzw. Speicheroperationen, die für Zugriffe auf Eingabe- und Ausgabe-Arrays sowie auf den Stack benötigt werden. Experimente [Lor04] haben gezeigt, daß sich Ganzzahl-Hilfsinstruktionen, die für die Berechnung der effektiven Adressen von Speicherzugriffen benötigten werden, oftmals negativ auf die Gesamtleistung numerischer Grundroutinen auswirken. Daher ist ein wesentlicher Fokus des Backends die heuristische Suche nach möglichst günstigen Sequenzen von Ganzzahl-Instruktionen für die Berechnung der effektiven Adressen.

Allokation von Gleitpunkt-Registern. Das MAP Backend führt die Registerallokation für alle nicht-ganzzahligen Register in einem Übersetzungsdurchgang aus und benutzt dazu die Farthest-First-Policy [Bel66, GGP04].

Anordnung von Instruktionen. Das MAP Backend beinhaltet eine Reihe verschiedener Code-Scheduler, die ein weites Scheduling-Spektrum – von domänenspezifischem High-Level-Scheduling bis zu Low-Level-Scheduling – abdecken. Zwei High-Level-Scheduler versuchen, den Registerdruck zu minimieren. Der Medium-Level-Scheduler ordnet Instruktionen *minimal* um, sodaß Instruktionslatenzen berücksichtigt werden, aber unnötige Code-Bewegung vermieden wird. Der Low-Level-Scheduler modelliert das Ausführungsverhalten des Zielprozessors und basiert auf dem List-Scheduling-Algorithmus [Muc97].

Abschätzung der Laufzeit. Das im Low-Level-Scheduler implementierte Prozessormodell verwendete Modell schätzt die Laufzeit von übersetztem Code ab. Das Ausführungsmodell des Low-Level-Schedulers berücksichtigt dabei (i) die Latenz von Instruktionen, (ii) den unterschiedlichen Durchsatz verschiedener Instruktionen, (iii) Einschränkungen seitens des Prozessers beim Dekodieren von Instruktionen, (iv) die benötigten Functional-Units und (v) etwaiges Forwarding von Registern.

4 Resultate

Mit Hilfe des MAP Compiler für IBM Blue Gene wurde eine Version of FFTW namens "FFTW-GEL for IBM Blue Gene" (BGL/FFTW-GEL) hergestellt. Diese für den in Blue Gene verwendeten IBM PowerPC 440 FP2 Prozessor optimierte Programmbibliothek wurde auf Korrektheit überprüft und bildet einen wesentlichen Grundbaustein naturwissenschaftlicher Anwendungssoftware auf IBM Blue Gene.

Abbildung 3 stellt die Leistung dar, die verschiedene Compiler auf IBM Blue Gene bei FFT-Signaltransformationen bestimmter Transformationslängen erreichen.

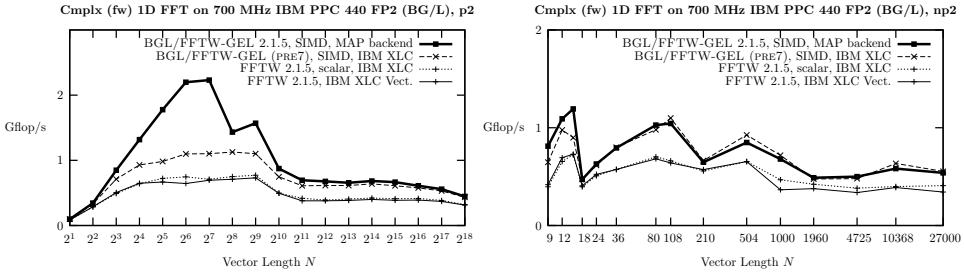

Abbildung 3: **Leistung von FFT-Codes auf einem IBM Blue Gene Prozessor.** Auf der linken Seite die Transformation von komplexen eindimensionalen Vektoren mit einer Länge 2^k, auf der rechten Seite Transformation mit aus mehreren Primfaktoren zusammengesetzten Vektorlängen verglichen. Der IBM PowerPC 440 FP2 Prozessor ist ein superskalarer 32-bit Prozessor, mit In-Order-Dual-Issue-Ausführung in einer siebenstufigen Pipeline, einer 2-weg SIMD Gleitpunktrecheneinheit, 32 kB L1-Cache, und 2 MB L3-Cache.

In Abbildung 3 werden folgende Compiler und Einstellungen miteinander verglichen: (i) "FFTW 2.1.5, scalar, IBM XL C" bezeichnet eine skalare (nicht-vektorisierte) Variante von FFTW, die vom IBM XL C Compiler übersetzt wurde. (ii) Bei "FFTW 2.1.5, IBM XL C Vect." wurden die automatische SIMD-Vektorisierung des IBM XL C Compilers aktiviert. (iii) "BGL/FFTW-GEL (Pre7), SIMD, IBM XL C" ist Variante von FFTW, bei der die numerischen Grundroutinen vom MAP Vectorizer und Optimizer in Intrinsic-Functions übersetzt wurden, und dann vom IBM XL C Compiler in Assembler-Code transformiert wurde. (iv) Bei "BGL/FFTW-GEL, SIMD, MAP backend" wurden die Grundroutinen gänzlich vom MAP Compiler – inklusive Backend – übersetzt.

Es zeigt sich der klare erkennbare Nutzen des MAP Vectorizers und Optimizers, und bei Transformationslängen 2^k eine deutliche Leistungssteigerung durch den Einsatz des MAP Backends. Bei aus mehreren Primfaktoren zusammengesetzten Transformationslängen ist das MAP Backend hingegen nur dann von Nutzen, wenn eine große Menge an flachen Grundroutinen für die jeweiligen Längen erzeugt wird, was aber aufgrund der resultierenden Zunahme der Codegröße nicht erfolgt ist. Bestenfalls – bei komplexen Transformationscodes der Länge 2^7 – erzeugt der MAP Compiler Code, der im Vergleich zu vom optimierenden IBM XL C Compiler erzeugten Code die dreifache Leistung bietet.

Literatur

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, Cambridge, UK, 1998.

[Bel66] L. A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5(2):pp. 78–101, July 1966.

[CT65] J. W. Cooley und J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19:pp. 297–301, 1965.

[Doc01] K. Dockser. Oedipus Architecture: Extensions to PowerPC BookE for Hummer^2. Bericht, IBM, August 2001.

[FJ98] M. Frigo und S. G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proceedings of ICASSP 1998 – IEEE International Conference on Acoustics Speech and Signal Processing*, Jgg. 3, Seiten 1381–1384, 1998.

[FJ05] M. Frigo und S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):pp. 216–231, 2005.

[FK01] M. Frigo und S. Kral. The Advanced FFT Program Generator GENFFT. AURORA Technical Report TR2001-03, Institute for Applied Mathematics and Numerical Analysis,Vienna University of Technology, 2001.

[FKLU05] F. Franchetti, S. Kral, J. Lorenz und C. W. Ueberhuber. Efficient Utilization of SIMD Extensions. *Proceedings of the IEEE*, 93(2):pp. 409–425, 2005.

[Fri99] M. Frigo. A fast Fourier transform compiler. *Proceedings of PLDI 1999 – ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seiten pp. 169–180, 1999.

[GB65] S. W. Golomb und L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12:pp. 516–524, 1965.

[GDS+06] F. Gygi, E. W. Draeger, M. Schulz, B. R. De Supinski, J. A. Gunnels, V. Austel, J. C. Sexton, F. Franchetti, S. Kral, C. W. Ueberhuber und J. Lorenz. Large-Scale Electronic Structure Calculations of High-Z Metals on the BlueGene/L Platform. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2006. Gordon Bell Prize winner.

[GGP04] J. Guo, M. Garzaran und D. Padua. The power of Belady's algorithm in register allocation for long basic blocks. In *Languages and Compilers for Parallel Computing*, Jgg. 2958 of *LNCS*, Seiten 374–390. Springer-Verlag, 2004.

[KFL+04] S. Kral, F. Franchetti, J. Lorenz, C. W. Ueberhuber und P. Wurzinger. FFT Compiler Techniques. In *Proceedings of CC'04 – 13th International Conference on Compiler Construction*, Jgg. 2985 of *LNCS*, Seiten 217–231. Springer-Verlag, 2004.

[KFLU03] S. Kral, F. Franchetti, J. Lorenz und C. W. Ueberhuber. SIMD Vectorization of Straight Line FFT Code. In *Proceedings of Euro-Par'03 – 9th International Conference on Parallel and Distributed Computing*, Jgg. 2790 of *LNCS*, Seiten 251–260. Springer-Verlag, 2003.

[KTU06] S. Kral, M. Triska und C. W. Ueberhuber. Compiler Technology for Blue Gene Systems. In *Proceedings of Euro-Par'06 – 12th International Conference on Parallel and Distributed Computing*, Jgg. 4128 of *LNCS*, Seiten 279–288. Springer-Verlag, 2006.

[LA00] S. Larsen und S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):pp. 145–156, 2000.

[LB00] R. Leupers und S. Bashford. Graph-based code selection techniques for embedded processors. *ACM Transactions on Design Automation of Electronic Systems*, 5(4):pp. 794–814, 2000.

[LKFU05] J. Lorenz, S. Kral, F. Franchetti und C. W. Ueberhuber. Vectorization techniques for the Blue Gene/L double FPU. *IBM Journal of Research and Development*, 49(2/3):pp. 437–446, 2005.

[Lor04] J. Lorenz. *Automatic SIMD Vectorization*. Ph. D. thesis, Institute for Analysis and Scientific Computing, Vienna University of Technology, 2004.

[MAA+05] J. E. Moreira, G. Almasi, C. Archer, R. Bellofatto, P. Bergner, J. R. Brunheroto, M. Brutman, J. G. Castanos, P. G. Crumley, M. Gupta, T. Inglett, D. Lieber, D. Limpert, P. McCarthy, M. Megerian, M. Mendell, M. Mundy, D. Reed, R. K. Sahoo, A. Sanomiya, R. Shok, B. Smith und G. G. Stewart. Blue Gene/L Programming and Operating Environment. *IBM Journal for Research and Development*, 49(2/3):pp. 367–376, 2005.

[MJJ+98] J. M. F. Moura, J. Johnson, R. W. Johnson, D. Padua, V. Prasanna, M. Püschel und M. M. Veloso. SPIRAL: Portable Library of Optimized Signal Processing Algorithms, 1998.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[PMJ+05] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson und N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2):pp. 232–275, 2005.

[WPD01] R. C. Whaley, A. Petitet und J. J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Parallel Computing*, 27:pp. 3–35, 2001.

[XJJP01] J. Xiong, J. Johnson, R. Johnson und D. Padua. SPL: A Language and Compiler for DSP Algorithms. In *Proceedings of PLDI 2001 – ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seiten 298–308, 2001.

[ZC91] H. Zima und B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1991.

Stefan Kral wurde am 29. Mai 1978 in Krems an der Donau (Niederösterreich) geboren. Nach Besuch der Volksschule "Pülslgasse" in Wien Liesing absolvierte er das Unterstufengymnasium sowie das humanistische Oberstufengymnasium "Franz Keimgasse" in Mödling bei Wien. Es folgten das Diplomstudium Informatik (Abschluß mit Auszeichnung im Jänner 2004) und das Doktoratsstudium der technischen Wissenschaften (Abschluß mit Auszeichnung im Juni 2006) an der Technischen Universität (TU) Wien. Von Oktober 1998 bis Jänner 2006 unterstützte er als TU-Studienassistent die Abhaltung der Lehrveranstaltung "logikorientierte Programmiersprachen". Von April 2004 bis März 2007 war er unter der Leitung von Prof. Christoph Überhuber als Forschungsassistent im Rahmen des High-Performance-Computing Spezialforschungsbereiches AURORA tätig. Mit seinen Beiträgen im Bereich Special-Purpose-Compilation für IBM Blue Gene war er Teil des Teams rund um François Gygi (LLNL), das 2005 und 2006 für den Gordon-Bell-Preis für Peak-Performance nominiert war und ihn 2006 gewann.